Beyond the good ol' LaunchAgents - 10 - Application script files

theevilbit.github.io/beyond/beyond_0010

April 2, 2021

> This is part 10 in the series of "Beyond the good ol' LaunchAgents", where I try to collect various persistence techniques for macOS. For more background check the introduction.

I started to explore to possibility of persisting on macOS through script files contained in an application. The basic idea is that if we find a script file, which is being executed by a given application, we can edit that script file, put our code inside, and wait for an execution. Such technique is highly dependent on the applications the user has installed, so I looked through first how rare / frequent is having such scripts inside applications. I started with the below searches:

```
find /Applications/ -name "*.sh"
find /Applications/ -name "*.py"
```

I have two Macs, one I use for private stuff and one for work, the second has significantly lower number of apps installed, only those I really need. First I checked my work MacBook as it has a much lower 'attack surface' and it turns out that these scripts files are not that rare at all, but they are also not everywhere. There is a common pattern of apps using scripts for specific purpose, at least based on the name, a few examples:

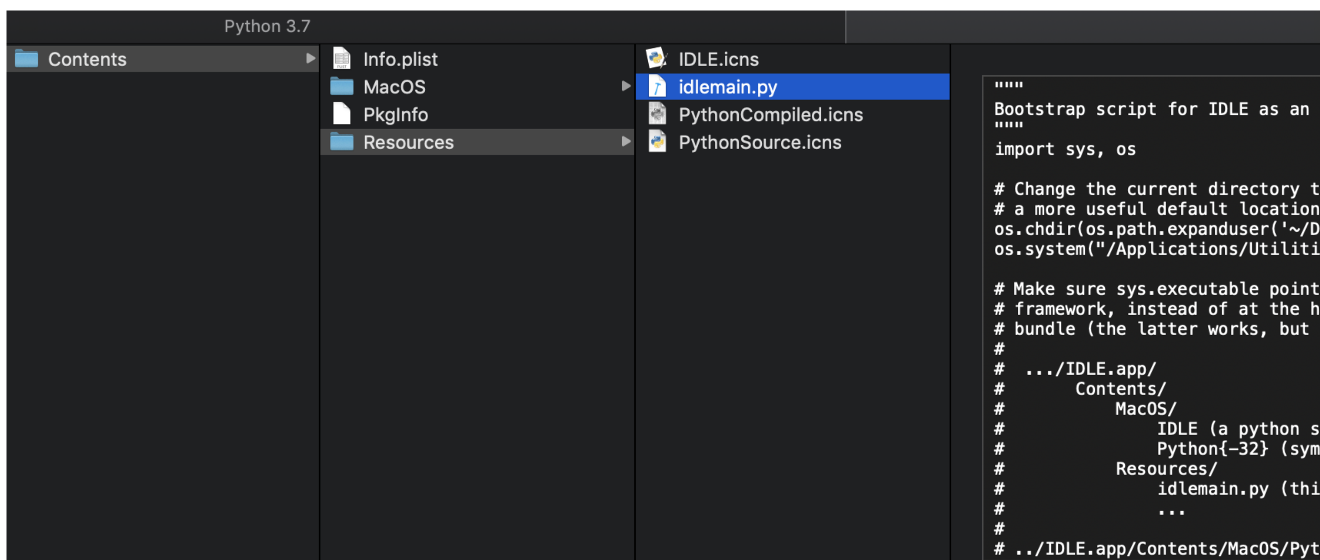
```
/Applications//Bear.app/Contents/Resources/Custom Tag
Keywords/keywordsplist_to_csv.py
/Applications//Hopper Disassembler v4.app/Contents/Resources/script_disassemble.py
/Applications//BBEdit.app/Contents/PlugIns/Language
Modules/Python.bblm/Contents/SharedSupport/py_check_syntax.py

/Applications//VMware Fusion.app/Contents/Library/shares/adduser.sh
/Applications//BBEdit.app/Contents/PlugIns/Language
Modules/ManPage.bblm/Contents/Resources/man2html.sh
```

Xcode is also full of such scripts. The problem with these that we don't know when they will be called, possibly it's not so frequent, so they are not ideal for persistence, as we want something that is always invoked when an applications starts. There is a chance that you can find a frequently run script somewhere, but those would require a check one by one, which I didn't do. You could also go about infecting every possible script you find, increasing the chances of being executed. The question is if there are any other scripts that will always run, and the answer is yes.

Python3 Idle.app

Although it's probably not that common people installing Python on a macOS system, as it's present by default, but if so, it contains the Idle.app editor. This application has a idlemain.py script in the resources folder, that is executed upon starting Idle. The app or the OS doesn't verify if the script was tampered with.



I have two problems with this, one is that finding this app somewhere is very unlikely, the second is that as you have to install this, the folder permissions are set for root access only. Even if you persist, you will only maintain yourself as the user and not as root. So it's not that ideal, but it works if really needed.

Sublime Text

This one is much more promising. This is a very popular text editor application, so you are likely to find it somewhere. Second, you install it by drag & drop to the application folder, so the user has the rights to edit the script file. If we edit the following file:

```
/Applications//Sublime\ Text.app/Contents/MacOS/sublime.py
```

It will be executed every time you start up Sublime Text. That's awesome! I used the following POC code:

```
import os
os.system("osascript -e 'Tell application \"System Events\" to display dialog \"
Message \"'")
```

You will get a prompt on Mojave to grant access, but you can add other code, which would do something else to avoid this prompt, and even if you get it, an average user will just grant access.

Sublime has a code signature and hash for this script, the reason macOS will not block tampering with the script, was covered by Thomas Reed in his talk about "Code Signing flaw in macOS": https://objectivebythesea.com/v1/talks/OBTS_v1_Reed.pdf In short macOS will

only verify code signature upon first execution, anything changes after that will be undetected.

GOG launchers

If I run the same search on my private MacBook there are a whole lot of other apps that has scripts inside, and I found one particularly funny. I used to buy games from https://gog.com I like that they are DRM free, I can backup the game, don't need to rely on online connectivity, etc… It turns out that they have very interesting script in one of the main packages to launch games:

```
#!/bin/bash
# GOG.com (www.gog.com)
# GOGLauncher Script

FIND_GAME=`find game/ -type d -maxdepth 1 -name "*.app"`
xattr -r -d com.apple.quarantine "$FIND_GAME"
open "${FIND_GAME}"
```

Stop for a minute and let the script sink in :) Here is what does it do in human language:

Upon starting the launcher script, it will look for apps in the game/ folder, remove the quarantine attribute and launch the app with the open command. The open command would invoke Gatekeeper, but only if the attribute is not removed! What this means, that you can replace the game-to-be-launched with anything you want (!!!) and it will be executed, even if it has a quarantine flag set. This is a signed launcher, which will launch code embedded within the app. BUT! There is always a BUT! It is not as good as it looks for first sight. It will only do this if the application was already run once, if not then the entire package, including the embedded game's file hashes will be verified, because there is a list of all files in the `Contents/_CodeSignature/CodeResources` file, which can't be altered, although this is an XML plist file as its hash is in the app's signature. See the above talk again. This means that you can't use this launcher to bypass Gatekeeper in a generic way. (Yes I spent some time to explore this option :)).

Some good resources I found on code signing: Inside Code Signing · objc.io http://www.newosxbook.com/articles/CodeSigning.pdf

This was a side track, but going back to original question, about using scripts, you can find many of those in GOG games, which you can expect users running, also they have a user permission set, so you can edit them.

The /Library folder

Another place to look for files is the `/Library` folder or the `~/Library` folder. The first one can contain exessive amount of files, so we might want to limit our search, where the user has write access to, using this syntax:

```
find /Library/ -name "*.sh" -perm +0200 -user username
find /Library/ -name "*.py" -perm -u+w -user username
```

There can be some gems as well, like this:

```
/Library//Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/
```

Here you can 'infect' many of the installed modules, so whenever they are imported you got code execution, you can expect these to be run more frequently.

Homebrew

This idea came from @philofishal. Homebrew is a very popular macOS package manager. Most powerusers have it installed. It's located at `/usr/local/bin/brew`.

```
% which brew
/usr/local/bin/brew

% file `which brew`
/usr/local/bin/brew: Bourne-Again shell script text executable, ASCII text
```

If we check the type of that file we can notice that it's a regular bash script file. As it's frequently run to install new packages or updates, it's an excellent target for backdoor and persistence.

Wireshark

Another idea that came from @philofishal. Wireshark installs a launchd file at `/Library/LaunchDaemons/org.wireshark.ChmodBPF.plist`, which means that it will be executed as root.

```
% cat /Library/LaunchDaemons/org.wireshark.ChmodBPF.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Label</key>
        <string>org.wireshark.ChmodBPF</string>
        <key>RunAtLoad</key>
        <true/>
        <key>Program</key>
        <string>/Library/Application Support/Wireshark/ChmodBPF/ChmodBPF</string>
</dict>
</plist>

% file "/Library/Application Support/Wireshark/ChmodBPF/ChmodBPF"
/Library/Application Support/Wireshark/ChmodBPF/ChmodBPF: Bourne-Again shell script
text executable, ASCII text
```

If we check the file, it's again a bash script, which we can easily backdoor if we got root level access. It can be hardly noticed.

Python import hijacking

If we talk about scripts, I have to mention this technique. It's well documented in many blog posts, talks, so won't go into too many details, but here is the TL;DR version: If you import a module in Python it will first look in the current folder for that PY script, and then on the other folders, where PYTHONPATH points to. We can get this list from Python, on my machine it looks like this:

```
Python 2.7.11 (v2.7.11:6d1b6a68f775, Dec  5 2015, 12:54:16)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> print sys.path
['', '/Library/Frameworks/Python.framework/Versions/2.7/lib/python27.zip',
'/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7',
'/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-darwin',
'/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac',
'/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac/lib-
scriptpackages',
'/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-tk',
'/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-old',
'/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-dynload',
'/Users/csaby/Library/Python/2.7/lib/python/site-packages',
'/usr/local/lib/python2.7/site-packages',
'/usr/local/Cellar/numpy/1.16.1/libexec/nose/lib/python2.7/site-packages',
'/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages',
'/Library/Python/2.7/site-packages',
'/System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python',
'/System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python/PyObjC']
```

Interestingly the 3rd path in the list `/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7` is writeable by the group where my account belongs to, so I can just persist there by injecting my code into one of the frequently used libraries, like `os` or `system`. The 2nd entry in the list `/Library/Frameworks/Python.framework/Versions/2.7/lib/python27.zip` doesn't exists so it's good to go.

Detection

I think this is something very hard to detect, especially if someone backdoors the Python libraries, however one potential solution would be to compare the timestamps of the files of a given application. They should be the same or at least pretty close. I did a few checks and the actual hour/minute value can differ, so you can go for the date. This is a quick and dirty python script to do that:

```
import os
import time
import sys

#original: https://stackoverflow.com/questions/17372696/pulling-files-and-timestamps-
from-a-directory-and-subdirectories
def find_timestamps(directory_path):
        all_files = []
        # Walk through files in directory_path, including subdirectories
        for root, _, filenames in os.walk(directory_path):
                for filename in filenames:
                        file_path   = root + '/' + filename
                        modified        = time.strftime('%Y-%m-%d',
time.localtime(os.path.getmtime(file_path)))
                        #time.ctime(os.path.getmtime(file_path))

                        # Process stuff for the file here, for example...
                        #print "File: %s" % file_path
                        #print "        Last modified: %s" % modified
                        all_files.append((file_path, modified))
        return all_files


def main():
        if (len(sys.argv) < 2):
                print "[-] no folder specified, useage: \n find_discrepancy.py
/path/to/app"
                sys.exit(-1)
        dir = sys.argv[1]
        if(not os.path.isdir(dir)):
                print "[-] This is not a directory"
                sys.exit(-1)
        all_files = find_timestamps(dir)
        t = ''
        for f in all_files:
                if t == '': #we set the first timestamp found as reference, could be
improved using stats
                        t = f[1] #the modified param is the second item in the tuple
                else:
                        if f[1]!= t:
                                print "[!] The file %s timestamp is different form
the rest, file: %s, rest: %s - potential backdoor" % (f[0], f[1], t)

if __name__== "__main__":
        main()
```

You could also monitor the hashes of all the files, and alert on changes. Here you will need to deal with updates, etc…

The same persistence idea is valid for Windows and Linux as well.