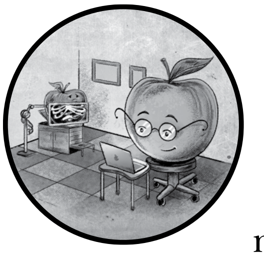
10

PERSISTENCE ENUMERATOR



In early 2014, a close friend begged me for help disinfecting his Mac. When I plopped myself in front of his screen, I saw obvious signs of a rampant adware infection: flagrant browser pop-ups, as well as a hijacked home page. Even worse, resetting his browser didn't work; it reverted to its infected state upon each reboot, suggesting the presence of a persistent component buried somewhere deep within the system.

At the time, I was an experienced Windows malware analyst just beginning my foray into the world of macOS. Naively, I thought I could download a tool capable of enumerating all persistent software installed on the system to reveal the malicious component. Well-known security tools, such as Microsoft's AutoRuns,[1] provided such a capability for Windows systems, but I soon discovered nothing similar existed for Macs.

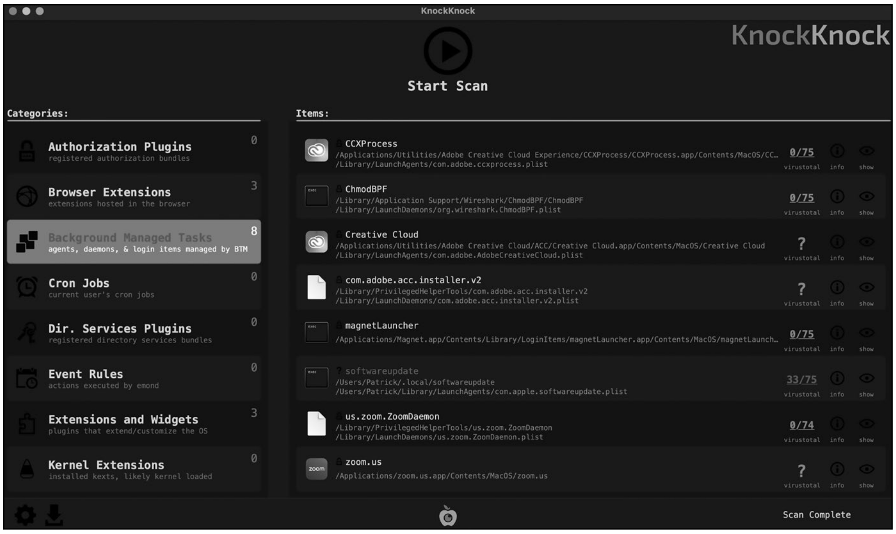
I returned home and spent the next few days putting together a Python script that, while embarrassingly ugly, was capable of enumerating several types of persistent software. Running the script revealed an unrecognized launch agent on my friend's computer that turned out to be the core persistent component of the adware. Once I removed it, his Mac was as good as new.

Realizing that my script could benefit other Mac users, I cleaned it up and released it under the moniker KnockKnock.[2] (Why KnockKnock? Because it tells you who's there!) Today, KnockKnock has evolved greatly from its beginnings as a humble command line script. Now distributed as a native macOS application, it's capable of detecting a myriad of persistently installed items on any macOS system. Coupled with an intuitive user interface (UI), integration with VirusTotal, and the ability to export its findings for ingestion into security information and event management (SIEM), it's the first tool I run on any Mac that I suspect is infected.

In this chapter, I'll walk through KnockKnock's design and implementation to give you an in-depth look at the tool and expand your understanding of the persistence methods that Mac malware often does (or could) abuse. In the process, we'll go beyond the detection mechanism discussed in Chapter 5, which focused solely upon the Background Task Management database, to look at other ways of persisting on macOS, including browser extensions and dynamic library hijacks. You can find the complete source code on Objective-See's GitHub page in the KnockKnock repository at *https://github.com/Objective-see/KnockKnock*.

Tool Design

KnockKnock is a standard UI-based application (as shown in Figure 10-1), but users can also execute it in the terminal as a command line tool.


KnockKnock
KnockKnock
Start Scan
Categories:
Authorization Plugins 0
registered authorization bundles
Browser Extensions 3
extensions hosted in the browser
Background Managed Tasks 8
agents, daemons, & login items managed by BTM
Cron Jobs 0
current user's cron jobs
Dir. Services Plugins 0
registered directory services bundles
Event Rules 0
actions executed by emond
Extensions and Widgets 3
plugins that extend/customize the OS
Kernel Extensions 0
installed kexts, likely kernel loaded
Items:
virustotal
info
show
Scan Complete

```
CCXProcess
/Applications/Utilities/Adobe Creative Cloud Experience/CCXProcess/CCXProcess.app/Contents/MacOS/CC…   0/75
/Library/LaunchAgents/com.adobe.ccxprocess.plist
ChmodBPF
/Library/Application Support/Wireshark/ChmodBPF/ChmodBPF                                              0/75
/Library/LaunchDaemons/org.wireshark.ChmodBPF.plist
Creative Cloud
/Applications/Utilities/Adobe Creative Cloud/ACC/Creative Cloud.app/Contents/MacOS/Creative Cloud      ?
/Library/LaunchAgents/com.adobe.AdobeCreativeCloud.plist
com.adobe.acc.installer.v2
/Library/PrivilegedHelperTools/com.adobe.acc.installer.v2                                              ?
/Library/LaunchDaemons/com.adobe.acc.installer.v2.plist
magnetLauncher
/Applications/Magnet.app/Contents/Library/LoginItems/magnetLauncher.app/Contents/MacOS/magnetLaunch…   0/75
softwareupdate
/Users/Patrick/.local/softwareupdate                                                                   33/75
/Users/Patrick/Library/LaunchAgents/com.apple.softwareupdate.plist
us.zoom.ZoomDaemon
/Library/PrivilegedHelperTools/us.zoom.ZoomDaemon                                                      0/74
/Library/LaunchDaemons/us.zoom.ZoomDaemon.plist
zoom.us
/Applications/zoom.us.app/Contents/MacOS/zoom.us                                                       ?
```

Figure 10-1: KnockKnock's user interface

As this isn't a book about writing UIs (thank goodness!), I won't delve into the code related to KnockKnock's UI. Instead, I focus mainly on its core components, such as its many plug-ins responsible for querying various aspects of the operating system to enumerate persistently installed items.

Command Line Options

The code for any Objective-C program starts at the standard `main` function, and KnockKnock is no exception. In its `main` function, KnockKnock begins by checking its program arguments to determine whether it should display its usage information or perform a command line scan (Listing 10-1).

```
int main(int argc, const char* argv[]) {
    ...
    if( (YES == [NSProcessInfo.processInfo.arguments containsObject:@"-h"]) ||
        (YES == [NSProcessInfo.processInfo.arguments containsObject:@"-help"]) ) {
        usage();
        goto bail;
    }

    if(YES == [NSProcessInfo.processInfo.arguments containsObject:@"-whosthere"]) {
        ...
        cmdlineScan();
    }
    ...
}
```

Listing 10-1: Parsing command line options

You might be familiar with accessing a program's command line arguments via the main function's `argv`. Objective-C supports this approach, but we can also access the arguments via the `arguments` array of the `processInfo` property in the `NSProcessInfo` class. This technique has several advantages, most notably that it converts the arguments into Objective-C objects. This means, for example, that we can use the `containsObject:` method to easily determine whether the user has specified a certain command line argument regardless of the order of the arguments.

To determine whether to run a command line scan, KnockKnock checks if the user specified the `-whosthere` command line option. If so, it invokes its `cmdlineScan` function to perform a scan of the system, printing out information about persistently installed items directly to the terminal.

Plug-ins

Because malware can persist on macOS in many ways and researchers discover new methods from time to time, KnockKnock's design relies on the concept of what I'll refer to as plug-ins. Each plug-in corresponds to one type of persistence and implements the logic to enumerate items of that persistence type. The plug-ins then call into other parts of KnockKnock to perform actions such as displaying each item in the UI. This modular approach provides a simple and efficient way to add support for new persistence techniques. For example, after the researcher Csaba Fitzl published the blog post "Beyond the Good Ol' LaunchAgents -32- Dock Tile Plugins," which detailed a new persistence strategy involving macOS Dock plug-ins,[3] I added a corresponding detection to KnockKnock via a new plug-in within the hour.

Each of KnockKnock's plug-ins inherits from a custom plug-in base class named PluginBase, which declares properties common to all plug-ins, as well as base methods. Found in *PluginBase.h*, it includes plug-in metadata, such as a name and a description, and arrays that the plug-in populates as it encounters persisting items (Listing 10-2).

```
@interface PluginBase : NSObject
    @property(retain, nonatomic)NSString* name;
    @property(retain, nonatomic)NSString* icon;
    @property(retain, nonatomic)NSString* description;

    @property(retain, nonatomic)NSMutableArray* allItems;
    @property(retain, nonatomic)NSMutableArray* flaggedItems;
    @property(retain, nonatomic)NSMutableArray* unknownItems;

    @property(copy, nonatomic) void (^callback)(ItemBase*);
    ....
@end
```

Listing 10-2: The base plug-in class's properties

The class also declares various base methods (Listing 10-3).

```
-(void)scan;
-(void)reset;
-(void)processItem:(ItemBase*)item;
```

Listing 10-3: The base plug-in class's methods

Each plug-in must implement the scan method with logic to enumerate one type of persistent item. For example, the Background Task Management plug-in will parse the Background Task Management database to extract persistent items managed by the Background Task Management subsystem, while the Browser Extension plug-in will enumerate installed browsers and, for each, extract any installed browser extensions. If researchers uncover a new persistence mechanism, we can trivially add a new plug-in with a scan method capable of enumerating items that persist in this new way.

The base class's scan method throws an exception if called directly (Listing 10-4).

```
@implementation PluginBase
...
-(void)scan {
    @throw [NSException exceptionWithName:kExceptName
    reason:[NSString stringWithFormat:kErrFormat, NSStringFromSelector(_cmd),
    [self class]] userInfo:nil];
}
@end
```

Listing 10-4: The base scan method will throw an exception if called.

This design allows KnockKnock to easily invoke each plug-in's `scan` method without having to know anything about how each plug-in actually enumerates persistent items of its specific type. The class provides base implementations for the other two methods, `reset` and `processItem:`, though plug-ins can override them if needed. (Otherwise, the plug-in will just call the base class's implementation.)

Both methods affect the application's UI. For example, when performing a UI scan, the `reset` method handles situations in which a user stops and then restarts a scan, while the `processItem:` method updates the UI as plug-ins uncover persistent items. During a command line scan, the `processItem:` method will still keep track of detected items and print each one to the terminal once the scan completes (Listing 10-5).

```
-(void)processItem:(ItemBase*)item {
    ...
    @synchronized(self.allItems) {
        [self.allItems addObject:item];
    }
}
```

Listing 10-5: Updating a global list of persistent items

KnockKnock declares a static list of all plug-ins by their class name. Later, the code iterates over this list, instantiating each plug-in (Listing 10-6).

```
static NSString* const SUPPORTED_PLUGINS[] = {@"AuthorizationPlugins",
@"BrowserExtensions", @"BTM", @"CronJobs", @"DirectoryServicesPlugins",
@"DockTiles", @"EventRules", @"Extensions", @"Kexts", @"LaunchItems",
@"DylibInserts", @"DylibProxies", @"LoginItems", @"LogInOutHooks",
@"PeriodicScripts", @"QuicklookPlugins", @"SpotlightImporters",
@"StartupScripts", @"SystemExtensions"};

PluginBase* pluginObj = nil;

for(NSUInteger i = 0; i < sizeof(SUPPORTED_PLUGINS)/sizeof(SUPPORTED_PLUGINS[0]); i++) {
    pluginObj = [[NSClassFromString(SUPPORTED_PLUGINS[i]) alloc] init]; ❶
    ...
}
```

Listing 10-6: Initializing each plug-in by name

For each plug-in class name, KnockKnock invokes the `NSClassFromString` API, which obtains a plug-in class based on the given name.[4] Then it invokes the class's `alloc` method to allocate an instance of the class (in other words, to create an object). Next, it invokes the newly created object's `init` method to allow the plug-in object to perform any initializations ❶. We'll consider some initialization examples shortly. Although not shown here, KnockKnock will then invoke each of the plug-in's `scan` methods.

Persistent Item Types

KnockKnock assigns one of three types to persistent items: file, command, or browser extension. Most persisted items are executable files, such as scripts or Mach-O binaries. However, as in the case of cron jobs, malware sometimes persists as a command; other times, it persists as a bundle of files and resources in the form of a browser extension. It's important for KnockKnock to correctly classify items, as each type has unique characteristics. For example, a persistent file might have extractable code signing information to help us classify it. We can also hash such files to check for known malware.

The three item types are subclasses of a custom `ItemBase` class, shown in Listing 10-7.

```
@interface ItemBase : NSObject
    @property(nonatomic, retain)PluginBase* plugin;

    @property BOOL isTrusted;
    @property(retain, nonatomic)NSString* name;
    @property(retain, nonatomic)NSString* path;
    @property(nonatomic, retain)NSDictionary* attributes;

    -(id)initWithParams:(NSDictionary*)params;
    -(NSString*)pathForFinder;
    -(NSString*)toJSON;
@end
```

Listing 10-7: The interface for the `ItemBase` class

This base class declares various properties, such as the plug-in that discovered the item, the item's name, and its path. Not all item types set every property. For example, commands don't have paths, whereas files and extensions do. The `ItemBase` class also implements base methods to initialize an item, return its path to show it in the Finder app, and convert it to JSON. Although objects that inherit from this base class can reimplement each method if they need to, the base class's implementation may suffice.

Once a plug-in's `scan` method completes, it stores any discovered items in a plug-in property called `allItems`. In a command line scan, KnockKnock converts each persistent item to JSON and appends it to a string that it prints out (Listing 10-8).

```
NSMutableString* output = [NSMutableString string];
...
for(NSUInteger i = 0; i < sizeof(SUPPORTED_PLUGINS)/sizeof(SUPPORTED_PLUGINS[0]); i++) {
    ...
    [plugin scan];

    for(ItemBase* item in plugin.allItems) {
        ...
        [output appendFormat:@"{%@},", [item toJSON]];
    }
    ...
}
```

Listing 10-8: Converting persistent items to JSON

Each item type implements its own logic to convert the information collected about a persistent item to JSON. Let's take a look at the implementation of the `toJSON` method for items whose type is `File` (Listing 10-9).

```
@implementation File
-(NSString*)toJSON {
    NSData* jsonData = nil;

    jsonData =
    [NSJSONSerialization dataWithJSONObject:self.signingInfo options:kNilOptions error:NULL]; ❶

    NSString* fileSigs =
    [[NSString alloc] initWithData:jsonData encoding:NSUTF8StringEncoding];

    jsonData =
    [NSJSONSerialization dataWithJSONObject:self.hashes options:kNilOptions error:NULL]; ❷

    NSString* fileHashes = [[NSString alloc] initWithData:jsonData encoding:
    NSUTF8StringEncoding];
    ...
}
```

Listing 10-9: Converting `File` object properties to JSON

First, the code makes use of the `NSJSONSerialization` class's `dataWithJSONObject:options:error:` method to convert various dictionaries into JSON. These dictionaries include the item's code signing information ❶ and hashes ❷. The method also converts numeric values from VirusTotal scan results (Listing 10-10).

```
NSString* vtDetectionRatio = [NSString stringWithFormat:@"%lu/%lu",
(unsigned long)[self.vtInfo[VT_RESULTS_POSITIVES] unsignedIntegerValue],
(unsigned long)[self.vtInfo[VT_RESULTS_TOTAL] unsignedIntegerValue]];
```

Listing 10-10: Computing a detection ratio based on scan results from VirusTotal

Technically, KnockKnock itself doesn't include logic to detect malicious code; it merely enumerates persistently installed items. This is by design, as it allows KnockKnock to detect new persistent malware even with no direct a priori knowledge of it. However, KnockKnock's integration with VirusTotal allows it to flag already known malware by submitting a POST request with a hash of each persistent item to a VirusTotal query API. This API returns basic detection information, such as how many antivirus engines scanned the items and how many of those engines flagged it as malicious. KnockKnock converts this data into a string ratio of the form *positive detections/antivirus engines* and then displays this result in the UI or command line output.[5]

The `toJSON` method finishes by building a single string object that combines the converted dictionaries, formatted numerical values, and all other properties of the item object (Listing 10-11).

```
NSString* json = [NSString stringWithFormat:@"\"name\": \"%@\", \"path\":
\"%@\", \"plist\": \"%@\", \"hashes\": %@, \"signature(s)\": %@, \"VT
detection\": \"%@\"", self.name, self.path, filePlist, fileHashes,
fileSigs, vtDetectionRatio];
```

Listing 10-11: Building a JSON-ified string

It returns this string to the caller to print out. For example, on a system infected with the persistent DazzleSpy malware, KnockKnock would display the following JSON in the terminal:

```
% KnockKnock.app/Contents/MacOS/KnockKnock -whosthere -pretty
{
    "path" : "\/Users\/User\/.local\/softwareupdate",
    "hashes" : {
        "md5" : "9DC9D317A9B63599BBC1CEBA6437226E",
        "sha1" : "EE0678E58868EBD6603CC2E06A134680D2012C1B"
    },
    "VT detection" : "35\/76",
    "name" : "softwareupdate",
    "plist" : "\/Library\/LaunchDaemons\/com.apple.softwareupdate.plist",
    "signature(s)" : {
        "signatureStatus" : -67062
    }
}
```

The output shows several red flags pointing to the fact that this item is likely malicious. For example, it's running from a hidden directory (*.local*), and while it claims to be an Apple software updater, its signature status is `-67062`, which maps to the `errSecCSUnsigned` constant. What conclusively identifies this item as malware, though, is the VirusTotal detection ratio, which shows that roughly half of the antivirus engines on the site flagged it as malicious.

Exploring the Plug-ins

KnockKnock has approximately 20 plug-ins to detect a myriad of persistent items, including items stored in Background Task Management, browser extensions, cron jobs, dynamic library inserts and proxies, kernel extensions, launch items, login items, Spotlight importers, system extensions, and many more. Although I won't cover every plug-in here, I'll dive into a few of them and provide examples of the malware they can detect.

Background Task Management

In Chapter 5, we explored the undocumented Background Task Management subsystem, which macOS leverages to govern and track persistent items such as launch agents, daemons, and login items. Through reverse engineering, I showed you how to deserialize the items managed by the subsystem, which could include persistently installed malware. We then created an open source library that I dubbed *DumpBTM*, which is available on GitHub (*https://github.com/objective-see/DumpBTM*). To enumerate persistently installed launch and login items, KnockKnock leverages this library.

NOTE *In Xcode, you can link in a library under your project's Build Phases tab. There, expand Link Binary With Libraries, click* +, *and then browse to the library.*

After linking in the *DumpBTM* library, KnockKnock's Background Task Management plug-in can directly invoke its exported APIs, such as its `parseBTM` function. The function takes a path to a Background Task Management file (or nil, to default to the system's file) and returns a dictionary containing deserialized metadata about each persistent item managed by Background Task Management. Listing 10-12 shows a snippet of the code in the plug-in's `scan` method.

```
#import "dumpBTM.h"

-(void)scan {
    ...
    if(@available(macOS 13, *)) {
        NSDictionary* contents = parseBTM(nil);
        ...
    }
}
```

Listing 10-12: Calling into the DumpBTM *library*

This code makes use of the `@available` Objective-C keyword to ensure that the plug-in executes only on versions 13 and newer of macOS (as the Background Task Management subsystem doesn't exist on earlier versions). KnockKnock then iterates over the metadata for each persistent item returned by the *DumpBTM* library's `parseBTM` function and, for each, instantiates a `File` item object. It does this by invoking the `File` class's `initWithParams:` method, which accepts a dictionary of values for the object, including a path and, for launch items, the property list.

Note that the code explicitly checks for a property list, as some persistent items in the Background Task Management database, such as login items, won't contain one (Listing 10-13). This is an important check, as inserting a nonexistent (nil) item into a dictionary will cause your program to crash.

```
NSMutableDictionary* parameters = [NSMutableDictionary dictionary];

parameters[KEY_RESULT_PATH] = item[KEY_BTM_ITEM_EXE_PATH];
```

```
if(nil != item[KEY_BTM_ITEM_PLIST_PATH]) {
    parameters[KEY_RESULT_PLIST] = item[KEY_BTM_ITEM_PLIST_PATH];
}

File* fileObj = [[File alloc] initWithParams:parameters];
```

Listing 10-13: Creating a dictionary of parameters to initialize a `File` object

With an initialized `File` object in hand, KnockKnock's Background Task Management plug-in can now invoke the base plug-in class's `processItem:` method to trigger a refresh of the UI or, in a command line scan, add the item to the list of items persistently installed on the system.

Using the *DumpBTM* library, KnockKnock can easily enumerate all persistent items managed by the subsystem. In the following output, you can see the tool displaying details of the cyber-espionage implant WindTail, which persists an app named *Final_Presentation.app* as a login item:

```
% KnockKnock.app/Contents/MacOS/KnockKnock -whosthere -pretty
...
"Background Managed Tasks" : [
    {
        "path" : "\/Users\/User\/Library\/Final_Presentation.app\/Contents\/MacOS\/usrnode",
        "hashes" : {
            "md5" : "C68A856EC8F4529147CE9FD3A77D7865",
            "sha1" : "758F10BD7C69BD2C0B38FD7D523A816DB4ADDD90"
        },
        "VT detection" : "41\/75",
        "name" : "usrnode",
        "plist" : "n\/a",
        "signature(s)" : {
            "signatureStatus" : -2147409652
        }
    }
]
```

Many antivirus engines on VirusTotal now flag the malware, and a check of its signature returns `-2147409652`, which maps to the "certificate revoked" constant, `CSSMERR_TP_CERT_REVOKED`. However, KnockKnock would have shown the presence of the persistent item even before the antivirus engines on VirusTotal developed signatures for it.

Unfortunately, no external library can enumerate many of KnockKnock's other classes of persistence, so we'll have to write more code ourselves. One example is the browser extension plug-in, which we'll look at now.

Browser Extension

Most macOS adware installs a malicious browser extension to hijack search results, display ads, or even intercept browser traffic. Common examples of such adware include Genieo, Yontoo, and Shlayer.

Because no macOS APIs can enumerate installed browser extensions, KnockKnock must do so itself. Worse, as each browser manages its extensions in its own way, KnockKnock must implement specific enumeration code for each. Currently, the tool supports extension enumeration for Safari, Chrome, Firefox, and Opera browsers. In this section, we'll cover the code specific to Safari.

To list the installed browsers, KnockKnock uses relatively unknown Launch Services APIs (Listing 10-14).

```
-(NSArray*)getInstalledBrowsers {
    NSMutableArray* browsers = [NSMutableArray array];
  ❶ CFArrayRef browserIDs = LSCopyAllHandlersForURLScheme(CFSTR("https"));

    for(NSString* browserID in (__bridge NSArray *)browserIDs) {
        CFURLRef browserURL = NULL;
      ❷ LSFindApplicationForInfo(kLSUnknownCreator,
        (__bridge CFStringRef)(browserID), NULL, NULL, &browserURL);

        [browsers addObject:[(__bridge NSURL *)browserURL path]];
        ...
    }
    ...
    return browsers;
}
```

Listing 10-14: Obtaining a list of installed browsers using Launch Services APIs

The code invokes the `LSCopyAllHandlersForURLScheme` API with the URL scheme `https` ❶, which returns an array containing the bundle IDs of applications capable of handling that scheme. The code then invokes the `LSFindApplicationForInfo` API to map each ID to an application path ❷, saving these into an array that it returns to the caller.

In macOS 12, Apple added the `URLsForApplicationsToOpenURL:` method to the `NSWorkspace` class to return all applications capable of opening a specified URL. Invoking this method with a URL to a web page will return a list of all installed browsers. For newer versions of macOS, KnockKnock makes use of this API (Listing 10-15).

```
#define PRODUCT_URL @"https://objective-see.org/products/knockknock.html"

NSMutableArray* browsers = [NSMutableArray array];
if(@available(macOS 12.0, *)) {
    for(NSURL* browser in [NSWorkspace.sharedWorkspace URLsForApplicationsToOpenURL:
    [NSURL URLWithString:PRODUCT_URL]]) {
        [browsers addObject:browser.path];
    }
}
```

Listing 10-15: Obtaining a list of installed browsers with the `URLsForApplicationsToOpenURL:` method

You can find the code to enumerate Safari browser extensions in the `scanExtensionsSafari:` method of KnockKnock's browser extension plug-in. In Listing 10-16, the code invokes this method with Safari's location, found using the previous code.

```
NSArray* installedBrowsers = [self getInstalledBrowsers];

for(NSString* installedBrowser in installedBrowsers) {
    if(NSNotFound != [installedBrowser rangeOfString:@"Safari.app"].location) {
        [self scanExtensionsSafari:installedBrowser];
    }
    ...
}
```

Listing 10-16: Invoking Safari-specific logic to enumerate its extensions

The location of Safari's browser extensions has changed over the years; you could find them in the *~/Library/Safari/Extensions* directory until Apple decided to move them into the keychain. Older versions of KnockKnock tried to keep up with these changes, but now, it uses a simpler method: executing the macOS `pluginkit` utility (Listing 10-17).

```
for(NSString* match in @[@"com.apple.Safari.extension", @"com.apple.Safari.content-blocker"]) {
    NSData* taskOutput = execTask(PLUGIN_KIT, @[@"-mAvv", @"-p", match]);
    ...
}
```

Listing 10-17: Enumerating installed Safari extensions

The `-m` argument finds all plug-ins that match the search criteria specified in the `-p` argument; the `-A` argument returns all versions of the installed plug-ins, rather than just the highest version; and `-vv` returns verbose output that includes the display name and parent bundle. For the `-p` argument, we first use `com.apple.Safari.extension`, then `com.apple.Safari.content-blocker`. This ensures that we enumerate both traditional extensions and content blocker extensions.

We execute `pluginkit` in a helper function we've named `execTask` (discussed in Chapter 1), which simply launches the specified program along with any specified arguments and returns the output to the caller. Try running `pluginkit` yourself to enumerate the Safari extensions installed on your Mac. In the following output, you can see that I've installed an ad blocker:

```
% pluginkit -mAvv -p com.apple.Safari.extension
...
org.adblockplus.adblockplussafarimac.AdblockPlusSafariToolbar
Path = /Applications/Adblock Plus.app/Contents/PlugIns/Adblock Plus Toolbar.appex
UUID = 87C62A05-974F-4E6C-81EE-304D4548DA60
SDK = com.apple.Safari.extension
Parent Bundle = /Applications/Adblock Plus.app
Display Name = ABP Control Panel
Short Name = $(PRODUCT_NAME)
Parent Name = Adblock Plus
Platform = macOS
```

Leveraging this external binary has the downside of introducing a dependency and the need to parse its output, but it's still the most reliable option. There are many ways to parse any output. In Listing 10-18, KnockKnock takes the approach of extracting each extension's name, path, and UUID.

```
-(void)parseSafariExtensions:(NSData*)extensions browserPath:(NSString*)browserPath {
    NSMutableDictionary* extensionInfo = [NSMutableDictionary dictionary];

    extensionInfo[KEY_RESULT_PLUGIN] = self;
    extensionInfo[KEY_EXTENSION_BROWSER] = browserPath;

    for(NSString* line in
    [[[NSString alloc] initWithData:extensions encoding:NSUTF8StringEncoding]
    componentsSeparatedByCharactersInSet:[NSCharacterSet newlineCharacterSet]]) {
        NSArray* components = [[line stringByTrimmingCharactersInSet:
        [NSCharacterSet whitespaceCharacterSet]] componentsSeparatedByString:@"="];
        // key and value set to first and last component

        if(YES == [key isEqualToString:@"Display Name"]) {
            extensionInfo[KEY_RESULT_NAME] = value;
        } else if(YES == [key isEqualToString:@"Path"]) {
            extensionInfo[KEY_RESULT_PATH] = value;
        } else if(YES == [key isEqualToString:@"UUID"]) {
            extensionInfo[KEY_EXTENSION_ID] = value;
        }
        ...
    }
}
```

Listing 10-18: Parsing output containing installed Safari extensions

The parsing code separates the output line by line, then splits each line into key-value pairs using an equal sign (=) as a delimiter. This will, for example, split the line `Path = /Applications/Adblock Plus.app/Contents/PlugIns/Adblock Plus Toolbar.appex` into the key `Path` and a value containing the path to the installed ad blocker extension. The code then extracts key-value pairs of interest, such as the path, name, and UUID.

Using the path to the extension, we load its *Info.plist* file and extract a description of the extension from the `NSHumanReadableDescription` key (Listing 10-19).

```
details = [NSDictionary dictionaryWithContentsOfFile:
[NSString stringWithFormat:@"%@/Contents/Info.plist",
extensionInfo[KEY_RESULT_PATH]]][@"NSHumanReadableDescription"];

extensionInfo[KEY_EXTENSION_DETAILS] = details;

Extension* extensionObj = [[Extension alloc] initWithParams:extensionInfo];
```

Listing 10-19: Initializing an `Extension` object for each extension

Finally, we create a KnockKnock browser `Extension` item object with the collected extension metadata.

Dynamic Library Insertion

A malware sample known as Flashback shattered the notion that Apple's operating system was immune to malware.[6] Flashback exploited an unpatched vulnerability capable of automatically infecting users who browsed to a malicious website. Discovered in 2012, it amassed more than half a million victims, making it the most successful Mac malware at the time.

Flashback also persisted in a novel and stealthy manner. On an infected system, the malware gained user-assisted persistence by subverting Safari's *Info.plist* file and inserting the following dictionary under a key named `LSEnvironment`:

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

The dictionary's `DYLD_INSERT_LIBRARIES` key contains a string pointing to the malicious library *UnHackMeBuild*. Safari will load this library into the browser when launched, where the malware could stealthily execute.

Today, Apple has mostly mitigated dylib insertions via the `DYLD_INSERT_LIBRARIES` environment variable and other approaches. The dynamic loader now ignores these variables in a wide range of cases, such as for platform binaries or for applications compiled with the hardened runtime.[7] However, programs supporting third-party plug-ins, especially on older versions of macOS, may still be at risk.

As such, KnockKnock contains a plug-in to detect this type of subversion. It scans launch items and applications, checking for the presence of a `DYLD_INSERT_LIBRARIES` entry. For launch items, this entry lives under the `EnvironmentVariables` key in their property list file, and for applications, you can find it under a key named `LSEnvironment` in the app's *Info.plist* file, as we saw with Flashback. Because legitimate items rarely make use of persistent `DYLD_INSERT_LIBRARIES` insertions, you should closely examine any that you uncover.

Other plug-ins require a similar list of all launch items and applications, so KnockKnock produces this list in a global enumerator. Let's briefly look at how KnockKnock tackles such enumeration, focusing on the case of installed apps, as there are multiple ways to list these items on a Mac. The least recommended is to manually enumerate bundles found in the common application directories (such as */Applications*), as you'd have to take into account subdirectories such as */Applications/Utilities/*, as well as user-specific applications. Plus, applications could be installed in other locations.

A Stack Overflow post suggests better options.[8] These include leveraging the `lsregister` utility to list all applications that have been registered with Launch Services, using the `mdfind` utility or related Spotlight APIs to list all applications indexed by macOS, or making use of the macOS system_profiler utility to obtain a list of applications known to the operating system's software configuration.

KnockKnock opts for the system_profiler approach. The tool can output XML or JSON, which is easy to programmatically ingest and parse. Here is an example of XML output, along with the metadata for an instance of KnockKnock installed on my computer:

```
% system_profiler SPApplicationsDataType -xml
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<array>
    <dict>
    ...
    <key>_items</key>
    <array>
        <dict>
             <key>_name</key>
             <string>KnockKnock</string>
             <key>arch_kind</key>
             <string>arch_arm_i64</string>
             ...
             <key>path</key>
             <string>/Applications/KnockKnock.app</string>
             <key>signed_by</key>
             <array>
                <string>Developer ID Application: Objective-See, LLC (VBG97UB4TA)</string>
                <string>Developer ID Certification Authority</string>
                <string>Apple Root CA</string>
             </array>
             <key>version</key>
             <string>2.5.0</string>
        </dict>
        ...
```

KnockKnock executes system_profiler via the execTask helper function discussed earlier in this chapter (Listing 10-20).

```
-(void)enumerateApplications {
    NSData* taskOutput = execTask(SYSTEM_PROFILER, @[@"SPApplicationsDataType", @"-xml"]); ❶

    NSArray* serializedOutput =
    [NSPropertyListSerialization propertyListWithData:taskOutput
    options:kNilOptions format:NULL error:NULL]; ❷

    self.applications = serializedOutput[0][@"_items"]; ❸
}
```

Listing 10-20: Installed applications enumerated via system_profiler

Once this helper function returns ❶, KnockKnock serializes the XML output into an Objective-C object ❷, then saves the list of applications found under the _items key into an instance variable aptly named applications ❸.

Now that KnockKnock's global enumerator has obtained a list of applications (and launch items, although I didn't show this logic here), the dylib insertion plug-in can scan each, looking for the addition of the DYLD_INSERT_LIBRARIES environment variable. Listing 10-21 shows this implementation in a method called scanApplications.

```
-(void)scanApplications {
    ...
    for(NSDictionary* installedApp in sharedItemEnumerator.applications) { ❶
        NSBundle* appBundle = [NSBundle bundleWithPath:installedApp[@"path"]]; ❷
        NSURL* appPlist = appBundle.infoDictionary[@"CFBundleInfoPlistURL"]; ❸
        NSDictionary* enviroVars = appBundle.infoDictionary[@"LSEnvironment"]; ❹

        if( (nil == enviroVars) ||
            (nil == enviroVars[@"DYLD_INSERT_LIBRARIES"]) ) {
            continue;
        }

        NSString* dylibPath = enviroVars[@"DYLD_INSERT_LIBRARIES"]; ❺

        File* fileObj = [[File alloc] initWithParams:
        @{KEY_RESULT_PLUGIN:self, KEY_RESULT_PATH:dylibPath, KEY_RESULT_PLIST:appPlist.path}];

        [super processItem:fileObj];
    }
}
```

Listing 10-21: Enumerating applications containing an inserted environment variable

The code iterates over all apps found by the global enumerator ❶. For each, it uses the application's path to load the application's bundle ❷, which has useful metadata about the application. This includes the contents of the app's *Info.plist* file, which we can access through the bundle object's infoDictionary property. After extracting the path to the *Info.plist* file ❸, it uses the key LSEnvironment to extract the dictionary containing specific environment variables ❹. Of course, most apps won't set any environment variables, so the code skips these. However, for those that have the DYLD_INSERT_LIBRARIES key set, the code extracts its value: a path to the library inserted each time the application is run ❺. In Flashback, which subverted Safari, recall that the key-value pair looks like this:

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

Finally, the code in the plug-in creates and processes a File item object representing the inserted library, saving it to the list of persistent items uncovered by KnockKnock to then print to the terminal or display in the UI.

Dynamic Library Proxying and Hijacking

The last plug-in I'll cover in this chapter detects two other persistence mechanisms that make use of dynamic libraries. *Dylib proxying* replaces a library on which a target process depends with a malicious library. Whenever the target application starts, the malicious dynamic library loads and runs as well. To keep the application from losing legitimate functionality, it proxies requests to and from the original library.[9]

Closely related to dylib proxying is *dylib hijacking*, which exploits the fact that the loader may look for dependencies in multiple locations. Malware could take advantage of this behavior by tricking the loader into using a malicious dependency instead of a legitimate one. Although malware doesn't commonly abuse this technique, the post-exploitation agent EmPyre supports it as a persistence mechanism.[10] Dynamic libraries that perform such hijacking also proxy requests to keep from breaking legitimate functionality.

To detect either technique, KnockKnock generates a list of dynamic libraries, then checks each for an `LC_REEXPORT_DYLIB` load command that loads and proxies requests to the original library. While this load command is legitimate, benign libraries rarely use it, so we should closely examine any that do.

Unfortunately, there isn't a simple way to list all dynamic libraries installed on a macOS system, so KnockKnock focuses on those that are currently open or loaded by running processes. This approach isn't as comprehensive as a scan of the entire system, but then again, any persisted malware is probably running somewhere.

To build a list of loaded libraries, KnockKnock runs the `lsof` utility to list all open files on the system, then filters out everything but executables. If a dynamic library has been loaded somewhere, there should be an open file handle to it, which `lsof` can enumerate.

While getting a list of open files is fairly simple, determining whether a file is executable isn't as easy as you might expect. You can't just look for files whose extension is *.dylib* because that list wouldn't include frameworks, which are technically libraries but don't normally end in *.dylib*. For example, take a look at the *Electron* framework. The `file` command reports that it is indeed a dynamic library, though its extension isn't *.dylib*:

```
% file "/Applications/Signal.app/Contents/Frameworks/Electron
Framework.framework/Electron Framework"
Mach-O 64-bit dynamically linked shared library arm64
```

Another strategy might be to check which of the open files are binaries by checking the file's executable bit, but this would include scripts and other random files on macOS, such as certain archives (which, as we can see here, have the executable bit, `x`, set):

```
% ls -l /System/Library/PrivateFrameworks/GPUCompiler.framework/Versions/
32023/Libraries/lib/clang/32023.26/lib/darwin/libair_rt_iosmac.rtlib
-rwxr-xr-x  1 root  wheel  140328 Oct 19 21:35
```

```
% file /System/Library/PrivateFrameworks/GPUCompiler.framework/Versions/
32023/Libraries/lib/clang/32023.26/lib/darwin/libair_rt_iosmac.rtlib
current ar archive
```

While you could manually parse each file, looking for a universal or Mach-O magic value, it turns out an Apple-provided API can do this for you. The relatively unknown `CFBundleCopyExecutableArchitecturesForURL` API extracts the executable architecture of a file, returning `NULL` or an empty array for nonbinary files.[11] KnockKnock, which makes use of this API, also checks for binaries of supported architectures (Listing 10-22).

```
BOOL isBinary(NSString* file) {
    static dispatch_once_t once;
    static NSMutableArray* supportedArchitectures = nil;

    dispatch_once(&once, ^ {
        supportedArchitectures = ❶
        [@[[NSNumber numberWithInt:kCFBundleExecutableArchitectureI386],
        [NSNumber numberWithInt:kCFBundleExecutableArchitectureX86_64]] mutableCopy];

        if(@available(macOS 11, *)) { ❷
            [supportedArchitectures addObject:
            [NSNumber numberWithInt:kCFBundleExecutableArchitectureARM64]];
        }
    });

    CFArrayRef architectures = CFBundleCopyExecutableArchitecturesForURL( ❸
    (__bridge CFURLRef)[NSURL fileURLWithPath:file]);

    NSNumber* matchedArchitecture = [(__bridge NSArray*)architectures
    firstObjectCommonWithArray:supportedArchitectures]; ❹
    ...
    return nil != matchedArchitecture;
}
```

Listing 10-22: Determining whether an item is a binary

The `isBinary` function builds an array of architectures with values for both 32 and 64 Intel in a `dispatch_once` to ensure that the initialization only occurs once, as we'll invoke this function for every file any process has open ❶. Also, the code makes use of the `@available` Objective-C keyword to only add the ARM64 architecture on versions of macOS that support it ❷.

Next, we extract the executable architecture of the passed-in file ❸, using the `firstObjectCommonWithArray:` method to check for any of the supported architectures ❹. If we find them, we can be sure that the open file is indeed a binary capable of executing on the macOS system. We add these binaries to a list of dynamic libraries that KnockKnock will shortly check for proxying capabilities.

KnockKnock also enumerates all running processes to extract the dependencies of the process's main binary. Each of these dependencies is added to the list of libraries to check (Listing 10-23).

```
-(NSMutableArray*)enumLinkedDylibs:(NSArray*)runningProcs {
    NSMutableArray* dylibs = [NSMutableArray array];

    for(NSString* runningProc in runningProcs) { ❶
        MachO* machoParser = [[MachO alloc] init]; ❷
        [machoParser parse:runningProc classify:NO];

        [dylibs addObjectsFromArray:machoParser.binaryInfo[KEY_LC_LOAD_DYLIBS]]; ❸
        [dylibs addObjectsFromArray:machoParser.binaryInfo[KEY_LC_LOAD_WEAK_DYLIBS]];
    }
    ...
    return [[NSSet setWithArray:dylibs] allObjects]; ❹
}
```

Listing 10-23: Enumerating the dependencies of all running processes

To enumerate all running processes, the plug-in makes use of the `proc_listallpids` API discussed in Chapter 1. Then, to extract each process's dependencies, it invokes a method named `enumLinkedDylibs`, which iterates over each loaded process ❶, parses it using a Mach-O class I wrote based on code in Chapter 2 ❷, and saves both strong and weak dependencies ❸. Finally, the function returns a list containing all dependencies found in all running processes ❹.

Next, we scan the list of libraries enumerated via `lsof` and via the running processes (Listing 10-24).

```
-(NSMutableArray*)findProxies:(NSMutableArray*)dylibs {
    NSMutableArray* proxies = [NSMutableArray array];

    for(NSString* dylib in dylibs) {
      ❶ MachO* machoParser = [[MachO alloc] init];
        [machoParser parse:dylib classify:NO];

      ❷ if(MH_DYLIB != [[machoParser.binaryInfo[KEY_MACHO_HEADERS]
        firstObject][KEY_HEADER_BINARY_TYPE] intValue]) {
            continue;
        }

      ❸ if([machoParser.binaryInfo[KEY_LC_REEXPORT_DYLIBS] count]) {
            [proxies addObject:dylib];
        }
    }
    return proxies;
}
```

Listing 10-24: Checking whether a binary is a dynamic library that (likely) performs proxying

For each library to scan, the code snippet parses it via the Mach-O class ❶. Specifically, it checks the type of binary, ignoring any that aren't explicitly dynamic libraries (identified by the `MH_DYLIB` type) ❷. For dynamic libraries, it checks and saves the library if it has a load command of type `LC_REEXPORT_DYLIB` ❸.

The method returns a list of any proxy libraries it finds so KnockKnock can display them to the user, either in the terminal or in the UI.

Conclusion

Most Mac malware persists, so a tool that can enumerate persistently installed items can uncover even sophisticated or never-before-seen threats. In this chapter, we examined KnockKnock, a tool that provides this capability, leaving persistent Mac malware with almost no hope of remaining undetected. In the next chapter, we'll explore persistence further and cover a tool capable of detecting persistent Mac malware in real time.

Notes

1. See *https://learn.microsoft.com/en-us/sysinternals/downloads/autoruns.*
2. See *https://web.archive.org/web/20180117193229/https://github.com/synack/knockknock.*
3. Csaba Fitzl, "Beyond the Good Ol' LaunchAgents -32- Dock Tile Plugins," *Theevilbit Blog*, September 28, 2023, *https://theevilbit.github.io/beyond/beyond_0032/.*
4. "NSClassFromString(_:)," Apple Developer Documentation, *https://developer.apple.com/documentation/foundation/1395135-nsclassfromstring.*
5. You can read more about programmatic integration with VirusTotal in the service's developer documentation at *https://docs.virustotal.com/reference/overview.*
6. Patrick Wardle, "Methods of Malware Persistence on Mac OS X," VirusBulletin, September 24, 2014, *https://www.virusbulletin.com/uploads/pdf/conference/vb2014/VB2014-Wardle.pdf.*
7. Patrick Wardle, *The Art of Mac Malware: The Guide to Analyzing Malicious Software*, Volume 1 (San Francisco: No Starch Press, 2022), 36.
8. "Enumerate All Installed Applications on OS X," Stack Overflow, *https://stackoverflow.com/questions/15164132/enumerate-all-installed-applications-on-os-x.*
9. Wardle, *The Art of Mac Malware*, 1:36–37.
10. See *https://github.com/EmpireProject/EmPyre/blob/master/lib/modules/persistence/osx/CreateHijacker.py.*
11. "CFBundleCopyExecutableArchitecturesForURL," Apple Developer Documentation, *https://developer.apple.com/documentation/corefoundation/1537108-cfbundlecopyexecutablearchitectu?language=objc.*