How to thwart application bundle manipulation on macOS

By susannah.matt@redcanary.com

Archived: 2026-04-05 14:51:30 UTC

In the world of macOS, apps are, for all intents and purposes, packaged as application bundles. Traditionally, application bundles have been very loosely defined—and one could manipulate them to subvert core macOS security controls. In this context, a malicious actor could gain capabilities like privilege escalation and defense evasion. In this article, we'll break down two forms of application bundle manipulation:

- trojanizing legitimate application bundles for the purpose of subverting macOS security controls
- tampering with a legitimate application bundle on disk for the purpose of defense evasion and/or privilege escalation

What kind of apps are being targeted?

Subversion on this level has the potential to impact one of macOS's first layers of defense: Gatekeeper. In fact, this has become such a problem that developers have begun to point this out in Apple's own developer forums. In this thread, two developers discuss the inherent inability of macOS to protect their applications against tampering (as of Monterey). Unfortunately this has led to many developers' apps being pirated or abused.

Historically, legitimate apps have been the target of adversaries and software pirates alike. Software pirates are individuals who illicitly distribute applications like Microsoft Office, popular games (e.g., Fortnite), and productivity software (e.g, Adobe Photoshop) free of charge and usually over peer-to-peer services. Additionally, if a user downloads a pirated application that has also been trojanized, they run the risk of infecting themselves. Unfortunately, this is not an uncommon scenario, as seen with the EvilQuest/ThiefQuest, Shlayer and Silver Toucan/UpdateAgent/WizardUpdate malware families (just to name a few), which frequently leverage masquerading tactics to hide their true nature.

Pirated and trojanized applications frequently abuse legitimate application bundles to subvert core macOS security controls. However, they do not *explicitly* exhibit the characteristics of adversarial on-disk application bundle manipulation. For that, we'll need to take a look at a slightly more advanced malware family called XCSSET. But first, let's examine some of macOS's native security controls.

Relevant macOS security controls

Briefly, some relevant core security controls to mention here are:

- **System Integrity Protection** (SIP) is designed to restrict modification of protected system files and directories from the root user. Only entitled macOS processes have the privilege to modify them. This is essentially Apple's in-house version of Security Enhanced Linux (SELinux). This is the first of two runtime protection mechanisms enforced by macOS.

- **Hardened Runtime**, the second runtime protection mechanism, helps protect apps against attacks like code injection and memory space tampering. This is a capability applied by developers when building their apps.
- **Codesigning/Notarization** determines if an application is who it says it is and has been checked by Apple to be free from malware. Note though that these are both independent procedures.
- **Gatekeeper** validates the integrity of notarized application launches (through static analysis, codesigning, etc), protects against tampering, and can launch applications from a randomized path (called Gatekeeper Path Randomization (GPR) / App Translocation) to prevent malicious plugins from loading. Additionally, in macOS 13, Gatekeeper will also help prevent unauthorized app tampering (see our Gatekeeper overview below).
- **XProtect** comprises YARA signature-based malware detection (and remediation with the advent of XProtect Remediator in macOS 13). XProtect runs a scan when:
    - an app is opened for the first time
    - an app been modified on disk
    - XProtect signatures have been updated
- **Transparency, Consent, and Control** (TCC) are privacy protections controlling access to things like full disk access, automation, microphone, camera, etc.

What's the problem?

Prior to macOS 13 Ventura, the integrity of applications launched by the user was not always validated. In summary, the following checks would occur when a user launched a notarized application for the first time:

- **Was the app downloaded from the internet?** If so, a Gatekeeper first launch prompt will be shown and the user will be alerted that the application was downloaded from the internet. In practice, this means that macOS has "quarantined" the app. Side note: This is an **opt-in process for developers.** In other words, a file downloaded by any given app is only quarantined if the app's developer has opted into Launch Services File Quarantine Enabled (`LSFileQuarantineEnabled`).
- **Is the app validly signed?** This check is done to validate the application is who it says it is and ensure it's free of tampering.
- **Is the app safe/free from known malware?** This check is performed by XProtect YARA scanning.
    - If so, Gatekeeper will allow the application to launch.
    - Otherwise, Gatekeeper will block execution of the application only on first launch.

The crux of the problem here is that app integrity is only being validated on first launch and there are no additional security controls to protect against tampering. This means that an adversary could potentially design malware to abuse the gap in application integrity validation, as exhibited by the XCSSET malware family. Apple has since addressed this issue by enabling Ventura to deny unauthorized app "updates" and tampering requests. This feature was previewed and discussed in the WWDC22 session: What's new in privacy (see our Gatekeeper overview below).

Which malware families manipulate application bundles?

The following malware families manipulate application bundles in some form. This is not an exhaustive list:

- **XCSSET** drops a malicious applet into a legitimate app's (known as the donor) `.../Contents/MacOS/` directory. This is an explicit case of an adversary leveraging on-disk application bundle manipulation.
- **Shlayer*** has been observed armed with the Gatekeeper bypass: CVE-2021–30657. The **elegant logic flaw** here discovered by Cedric Owens enabled an adversary to bypass Gatekeeper simply by utilizing an executable script in the place of a Mach-O within an application bundle.
- **EvilQuest / Silver Toucan*** are examples of application bundles which have been cracked/modified and can masquerade as popular macOS apps like: Ableton Live, Little Snitch, and MacSnipper. Silver Toucan has been observed bypassing Gatekeeper.

* Please note, the activity observed from these families does not explicitly tie them into our definition of adversarial on-disk application bundle manipulation. However, prior to landing on a Mac they come with a modified installer, application bundle, or payload for the purpose of bypassing macOS security controls. This is somewhat expected behavior of cracked/modified applications like EvilQuest and Silver Toucan.

Why is this a potentially attractive path for adversaries?

Because modern macOS supports a wide array of avenues for code execution, the current security controls in place may have more work on their plates then they previously realized. Apps are just one way to get code to execute on macOS; scripts and Java archives can also get the job done. Importantly, Apple does not currently notarize all forms of executables. Supported deliverables are: apps, (non-app bundles: Frameworks and Loadables), UDIF Disk Images, and Flat installer packages. Unlike application bundles and Mach-O executables, security controls such as entitlements, codesigning, and hardened runtime do not apply to shell scripts because they do not make sense in this context. Entitlements are only able to be added to executables; this means that the shared libraries, frameworks, and in-process plug-ins inherit all the entitlements of the hosting executable.

EvilQuest's authors have been known to run scripts as part of an app install. The types of scripts you're able to run depends on the type of installer package: bundle or flat installer. This translates to preflight/install and postflight/install scripts.

The Gatekeeper logic flaw we mentioned earlier was enabled by Gatekeeper insufficiently validating the structure and content of the application bundle (appearing with a script executable instead of a more traditional Mach-O binary). Tricking macOS into executing code within the context of a legitimate app is the name of the game for an adversary here. Why? It could enable some nice capabilities:

- **Defense evasion:** The ability of a malicious program to hide its activity. In this case, malware like XCSSET latches onto a legitimate donor app.
- **Privilege escalation:** The malicious program gains elevated permissions to, for example, record or capture the screen, steal information from targeted apps, modify software updates, etc. This capability was also found in XCSSET.

What's the solution?

macOS needs a way to ensure that an application is who it says it is, all the time. Gatekeeper is one of the core security controls associated with ensuring an application has not been compromised. Below we will take a brief look at what Gatekeeper is and the upcoming changes in macOS 13 you should be aware of.

Gatekeeper

A key layer of the macOS security model, Gatekeeper's job is to only allow trusted application launches. It does this on the first launch of notarized apps through validating the requesting app's cryptographic signature and a notarization ticket, if one exists. Notarization tickets can be "stapled" to the app and/or stored on Apple's internet accessible servers. In System Settings (or via mobile device management with additional options), Gatekeeper's security policy "Allow apps downloaded from" can be modified to take on the following values:

- "App Store"
- "App Store and identified developers" (notarized applications)

Additionally, the `/usr/sbin/spctl` "security policy control" command-line tool can be used to modify the behavior of Gatekeeper, exposing an additional policy:

- `sudo spctl --global-disable` results in a Gatekeeper security policy allowing applications downloaded from: "Anywhere"

In practice, this means tagging applications downloaded from the internet with the `com.apple.quarantine` extended attribute (i.e., using a macOS application and not a command-line tool like `curl` or `wget`). As mentioned above, developers opt in to the file quarantine policy by by adding the following key to the `Info.plist`: `LSFileQuarantineEnabled`

Gatekeeper in macOS 13

Excluding the App Store case, prior to macOS 13 Ventura, Gatekeeper only validated the integrity of newly downloaded (i.e., quarantined) apps on first launch. However, starting with macOS 13, as explained during WWDC22 session: What's new in privacy, Gatekeeper will validate the integrity of all notarized apps on first launch (ensuring code signatures stay valid) Previously, Gatekeeper only validated the integrity of notarized apps that had been quarantined on first launch.

Additionally, Gatekeeper will attempt to **block unauthorized tampering** attempts alerting the user in the process. Modifying an app's executable, secondary resources, or the `Info.plist` are all common tasks that app updates might perform if authorized. This is a rather elegant solution to the expensive signature validation process that occurs on app first launch. By providing the user with the option to allow or deny incoming tampering requests for a given notarized app, the user can effectively prevent their notarized apps from being tampered with.

Developers with the same team identifier can still update their own apps with no problem. However, if another development team needs to be able to update that app, they would have to add an `NSUpdateSecurityPolicy` to the `Info.plist` and specify the `AllowProcesses` dictionary that maps a team identifier to a set of signing identifiers.

Image courtesy of developer.apple.com

How has on-disk application bundle manipulation been seen in the wild?

XCSSET achieves defense evasion and privilege escalation by living off of a donor application like Discord or Skype. XCSSET accomplishes these goals with help from the Open Scripting Architecture (OSA) that's baked into macOS. AppleScript—and by extension OSA—has traditionally been referred to as the "PowerShell of macOS." The execution of a supported scripting language through OSA enables some unique capabilities such as:

- access to core macOS APIs (e.g., **Foundation**, **Cocoa**)
- the potential to evade detection with execute-only scripts
- establishing persistence by writing launch agents / daemons
- hamstring macOS security controls (e.g., Gatekeeper and system updates)

These capabilities are enabled by the AppleScriptObjC/PyObjC bridge. What are the consequences of these bridges from a security perspective? Well, for one, Hardened Runtime (which helps protect against code injection and is required for notarization) would need to be disabled and, secondly, a JIT-compilation / "allow unsigned executable memory" entitlement would need to be added. Again, these requirements are only applicable to apps, disk images, and installers—the deliverables that can be notarized, as noted earlier.

Although these capabilities may seem a little scary on the surface, they also serve as a very real solution to an organizational management problem. For example, it'd be useful for Mac administrators to get or set management settings with a script, and this is where the AppleScriptObjC/PyObjC bridges come into play.

A note on endpoint security

- The **Endpoint Security Framework** (ESF), in its current implementation, does not include any **notify events** surrounding the Apple Event Manager (e.g., the sending / receiving of **Apple Events**). This means that any endpoint, detection, and response (EDR) sensor that wants to provide visibility into the sending or receiving of Apple Events would need to do so without the help of ESF, a much trickier job.
- In Apple's latest iteration of the ESF, we received a host of **new events** surrounding login items, authentication, OpenSSH, screen sharing, and XProtect (malware detection and remediation). Many of these notify events are useful in detecting the malware we've discussed so far.

XCSSET: a case study

We will now walk through how XCSSET, a recent malware family, explicitly demonstrates the concept of on-disk application bundle manipulation. In short, XCSSET drops a malicious applet in the donor application's `.../Contents/MacOS/` directory. Thus, complexities aside, the malicious applet inherits the privileges associated with the donor application and runs as a seemingly legitimate application. This has the added benefit of helping evade defenses as well.

Background

XCSSET, first reported on in August of 2020 by Trend Micro's Mac Threat Response team, is an extremely capable macOS malware family primarily targeting developers through malicious Xcode projects (an IDE designed explicitly for Apple platform development). Users are infected by XCSSET by downloading and building a compromised Xcode project, which subsequently downloads malicious applets. XCSSET contains multiple modules to do things like steal data and includes a replicator component to infect other Xcode projects.

Applets

Applets, for all intents and purposes, are apps. Their structure follows the same anatomy as generic application bundles and can be constructed in a few ways:

- graphically, with `Script Editor.app` by selecting the export to application option
- graphically, with `Automator.app` by selecting the application template when creating a new project. Automator describes applets as: self-running workflows. For example, one can make a simple macOS applet which starts the screensaver.
- via command line, with `osacompile` by specifying the `[-o]` with a `.app` file extension or `[-s]` option (for Stay-open applets).
- via **Foundation Library**, using the **NSAppleScript** API

Each of these methods will generate a valid application bundle with a thin Mach-O wrapper for OSA script execution(s). However, the Automator option will include a specialty `document.wflow` file and will name the Mach-O wrapper Automator Application Stub. By contrast, the other three options will create a `PkgInfo` file, keep compiled versions of the scripts in the Resources directory, and will name the Mach-O wrapper applet by default. Like regular apps, applets can be launched with the Launch Services API (essentially Finder) and must adhere to all the same Transparency, Consent, and Control (TCC) restrictions.

Applet observations

- By default and if constructed with `Automator.app`, the Mach-O binary in the `../MacOS` directory will be named `Application Stub`
- The Automator Application Stub Mach-O template binary can be found in `/System/Library/CoreServices/` as a universal binary
- The Mach-O binary links to many of the same `dylibs` and frameworks as `/usr/bin/osascript`
    - By disassembling the binary, we can see that the `ScriptMonitor.app` is being called as a way to provide the user with a visual indicator that a script is running. In other words, when an applet is launched so is `ScriptMonitor.app.` This enables another potential avenue for detection.
- If built using Automator, the contents of the OSA script will be placed in the `document.wflow` file (XML based) under the `<key>Action Parameters</key>` tag. Otherwise, there will be a `../Resources/Scripts` directory, which will contain all the scripts for the applet
- During execution, an `osascript` command line will generally not be seen in this case, unless the `osascript` binary is explicitly invoked
    - Applets can also contain shell scripts used to run OSA code. This would be the explicit case in which the `osascript` binary would be executed, and the `osascript` command line would be visible, as would the `ScriptMonitor` process

Procedures

XCSSET has been observed utilizing the `curl` and `/usr/bin/osacompile` command-line tools to download and compile an execute-only AppleScript file (into an applet) respectively. XCSSET will then drop the malicious applet at the `⋯/Contents/MacOS/` directory of the donor application.

Legitimate apps targeted by XCSSET as donors have included (shown below in the example from Jamf Threat Labs): Google Chrome, Telegram, Apple Notes, etc.

In summary, each Applet “downloaded” by XCSSET is a module containing functionalities such as:

- Information stealing: Safari, Contacts, Opera, Evernote, Apple Notes, Skype, Telegram, QQ, WeChat, etc.
    - **T1119: Automated Collection** (Collection)
- Injecting JavaScript backdoors into websites via a universal cross-site scripting (UXSS) attack
    - **T1562: Impair Defenses**
    - **T1539: Steal Web Session Cookie**
- Stealing browser cookies protected by SIP over SCP (secure copy)
    - **T1048: Exfiltration Over Alternative Protocol** (Exfiltration)
    - **T1539: Steal Web Session Cookie** (Credential Access)
- Disabling Gatekeeper, macOS updates, etc.
    - **T1562: Impair Defenses**
- Capturing screenshots, etc.
    - **T1113: Screen Capture** (Collection)

Consider the following example from Jamf Threat Labs featuring the `createDonorApp()` AppleScript function:

File: `screen_sim.applescript`

Function: `createDonorApp()`

First, notice the in-memory loading of `screen.applescript` from some C2 domain (T1059.002: AppleScript). The AppleScript code in memory is “execute-only” compiled line by line into an applet and then “dropped” to the location of a temporary application bundle (T1027.004: Compile After Delivery). Finally, the applet is copied to the donor application’s `…/Contents/MacOS/` directory (T1036: Masquerading and T1554: Compromise Client Software Binary).

Procedural examples from Jamf Threat Labs and Trend Micro Research

Example 1

This excerpt from Jamf Threat Labs was taken from the file `screen_sim.applescript` and the function `verifyCapturePermissions()`.

Here we see AppleScipt code enumerating the different donor applications XCSSET supports via “plugins”/applets.

Example 2

This excerpt from Jamf Threat Labs was taken from the file `screen_sim.applescript` and the function `createDonorApp()`.

Here we see step one of the primary ties XCSSET has to the application bundle manipulation technique: Creating a fake donor application bundle (and applet) with `osacompile -x -e & payload &-o & quoted form of tempApp`

(T1059.002: AppleScript, T1027.004: Compile After Delivery, and T1036: Masquerading).

Example 3

This excerpt from Jamf Threat Labs was also taken from the file `screen_sim.applescript` and the function `createDonorApp()`.

Here, XCSSET replaces the donor application bundle's Mach-O with the malicious applet plugin (T1059.002: AppleScript,
T1036: Masquerading, and T1554: Compromise Client Software Binary).

Example 4

As seen this example from Trend Micro Research, XCSSET is also able to disable software updates and Gatekeeper (T1059.002: AppleScript and T1562.001: Impair Defenses: Disable or Modify Tools), which is shown with the following AppleScript:

```
do shell script "spctl --master-disable" with administrator privileges
```

Sample XCSSET detection analytics

The following pseudo-detectors rely on telemetry from ESF and should help security teams detect XCSSET.

Detecting on the Xcode build process

This analytic uses `ES_EVENT_TYPE_NOTIFY_EXEC` and `ES_EVENT_TYPE_NOTIFY_FORK` events. It looks for the Xcode build process spawning shell scripts that, in turn, create a hex dump.

parent_process == `xcbbuildservice`

&&

process == (`bash` || `zsh` || or `sh`)

&&

child_processes == `/usr/bin/xxd`

Detecting behavioral process execution from within the GameKit cache directory

This analytics uses a single `ES_EVENT_TYPE_NOTIFY_EXEC` event, and looks explicitly for processes executing from the `.../library/caches/gamekit/` file directory.

process == [anything]

&&

file_path == `.../library/caches/gamekit/`

Detecting strange `applet /` command-line OSA behavior

This analytics relies on a single `ES_EVENT_TYPE_NOTIFY_EXEC` event and looks for suspicious OSA commands.

parent_process == (`applet` || `osascript`)

&&

command_line_includes (`osacompile`)

Detecting the in-memory downloading and compiling of payload applets

This analytics uses a single `ES_EVENT_TYPE_NOTIFY_EXEC` event and looks for the the execution of `curl` , `|` , or `osacompile` commands.

command_line_includes (`osacompile` && `|` && `curl`)

Source: https://redcanary.com/blog/mac-application-bundles/