BSD Privilege Escalation on macOS

This thread has been locked by a moderator.

315

This week I'm handling a DTS incident from a developer who wants to escalate privileges in their app. This is a tricky problem. Over the years I've explained aspects of this both here on DevForums and in numerous DTS incidents. Rather than do that again, I figured I'd collect my thoughts into one place and share them here.

If you have questions or comments, please start a new thread with an appropriate tag (*Service Management* or *XPC* are the most likely candidates here).

Share and Enjoy
—
Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

BSD Privilege Escalation on macOS

macOS has multiple privilege models. Some of these were inherited from its ancestor platforms. For example, Mach messages has a capability-based privilege model. Others were introduced by Apple to address specific user scenarios. For example, macOS 10.14 and later have mandatory access control (MAC), as discussed in On File System Permissions.

One of the most important privilege models is the one inherited from BSD. This is the classic users and groups model. Many subsystems within macOS, especially those with a BSD heritage, use this model. For example, a packet tracing tool must open a BPF device, `/dev/bpf*`, and that requires root privileges. Specifically, the process that calls `open` must have an effective user ID of 0, that is, the `root` user. That process is said to be *running as root*, and escalating BSD privileges is the act of getting code to run as root.

IMPORTANT Escalating privileges does not bypass all privilege restrictions. For example, MAC applies to all processes, including those running as root. Indeed, running as root can make things harder because TCC will not display UI when a `launchd` daemon trips over a MAC restriction.

Escalating privileges on macOS is not straightforward. There are many different ways to do this, each with its own pros and cons. The best approach depends on your specific circumstances.

Note If you find operations where a root privilege restriction doesn't make sense, feel free to file a bug requesting that it be lifted. This is not without precedent. For example, in macOS 10.2 (yes, back in 2002!) we made it possible to implement ICMP (ping) without root privileges. And in macOS 10.14 we removed the restriction on binding to low-number ports (r. 17427890). Nice!

Decide on One-Shot vs Ongoing Privileges

To start, decide whether you want one-shot or ongoing privileges. For one-shot privileges, the user authorises the operation, you perform it, and that's that. For example, if you're creating an un-installer for your product, one-shot privileges make sense because, once it's done, your code is no longer present on the user's system.

In contrast, for ongoing privileges the user authorises the installation of a `launchd` daemon. This code always runs as root and thus can perform privileged operations at any time.

Folks often ask for one-shot privileges but really need ongoing privileges. A classic example of this is a custom installer. In many cases installation isn't a one-shot operation. Rather, the installer includes a software update mechanism that needs ongoing privileges. If that's the case, there's no point dealing with one-shot privileges at all. Just get ongoing privileges and treat your initial operation as a special case within that.

Keep in mind that you can convert one-shot privileges to ongoing privileges by installing a `launchd` daemon.

Just Because You Can, Doesn't Mean You Should

Ongoing privileges represent an obvious security risk. Your daemon *can* perform an operation, but how does it know whether it *should* perform that operation?

There are two common ways to authorise operations:

- Authorise the user
- Authorise the client

To authorise the user, use Authorization Services. For a specific example of this, look at the EvenBetterAuthorizationSample sample code.

Note This sample hasn't been updated in a while (sorry!) and it's ironic that one of the things it demonstrates, opening a low-number port, no longer requires root privileges. However, the core concepts demonstrated by the sample are still valid.

The packet trace example from above is a situation where authorising the user with Authorization Services makes perfect sense. By default you might want your privileged helper tool to allow any user to run a packet trace. However, your code might be running on a Mac in a managed environment, where the site admin wants to restrict this to just admin users, or just a specific group of users. A custom authorisation right gives the site admin the flexibility to configure authorisation exactly as they want.

Authorising the client is a relatively new idea. It assumes that some process is using XPC to request that the daemon perform a privileged operation. In that case, the daemon can use XPC facilities to ensure that only certain processes can make such a request.

Doing this securely is a challenge. For specific API advice, see this post.

WARNING This authorisation is based on the code signature of the process's main executable. If the process loads plug-ins [1], the daemon can't tell the difference between a request coming from the main executable and a request coming from a plug-in.

[1] I'm talking in-process plug-ins here. Plug-ins that run in their own process, such as those managed by ExtensionKit, aren't a concern.

Choose an Approach

There are (at least) seven different ways to run with root privileges on macOS:

- A setuid-root executable
- The `sudo` command
- AppleScript's `do shell script` command, passing true to the `administrator privileges` parameter
- The long-deprecated `AuthorizationExecuteWithPrivileges` function, in Security framework
- The `SMJobBless` function, in Service Management framework
- An installer package (`.pkg`)
- The new-in-macOS 13 (currently in beta) `SMAppService` facility, a much-needed enhancement to the Service Management framework

To choose between them:

1. Do not use a setuid-root executable. Ever. It's that simple! Doing that is creating a security vulnerability looking for an attacker to exploit it.
2. If you're working interactively on the command line, use `sudo`.
   IMPORTANT `sudo` is not appropriate to use as an API. While it may be possible to make this work under some circumstances, by the time you're done you'll have code that's way more complicated than the alternatives.
3. If you're building an ad hoc solution to distribute to a limited audience, and you need one-shot privileges, use either `AuthorizationExecuteWithPrivileges` or AppleScript.
   While `AuthorizationExecuteWithPrivileges` still works, it's been deprecated for many years. Do not use it in a widely distributed product.
   The AppleScript approach works great from AppleScript, but you can also use it from native code using `NSAppleScript`. See the code snippet later in this post.
4. If you only need escalated privileges to install your product, consider using an installer package. That's *by far* the easiest solution to this problem.
   Keep in mind that an installer package can install a `launchd` daemon and thereby gain ongoing privileges.
5. If you need ongoing privileges but don't want to ship an installer package, use `SMAppService`. If you need to deploy to older systems, use `SMJobBless`.
   For instructions on using `SMAppService`, see Updating helper executables from earlier versions of macOS.
   For a comprehensive example of how to use `SMJobBless`, see the EvenBetterAuthorizationSample sample code. For the simplest possible example, see the SMJobBless sample code. That has a Python script to help you debug your setup. Unfortunately this hasn't been updated in a while; see this thread for more.

Hints and Tips

I'm sure I'll think of more of these as time goes by but, for the moment, let's start with the big one…

Do not run GUI code as root. In some cases you can make this work but it's not supported. Moreover, it's not *safe*. The GUI frameworks are huge, and thus have a huge attack surface. If you run GUI code as root, you are opening yourself up to security vulnerabilities.

Appendix: Running an AppleScript from Native Code

Below is an example of running a shell script with elevated privileges using `NSAppleScript`.

WARNING This is not meant to be the final word in privilege escalation. Before using this, work through the steps above to see if it's the right option for you.

Hint It probably isn't!

```
let url: URL = … file URL for the script to execute …

let script = NSAppleScript(source: """
    on open (filePath)
        if class of filePath is not text then
            error "Expected a single file path argument."
        end if
        set shellScript to "exec " & quoted form of filePath
        do shell script shellScript with administrator privileges
    end open
    """)!

// Create the Apple event.

let event = NSAppleEventDescriptor(
    eventClass: AEEventClass(kCoreEventClass),
    eventID: AEEventID(kAEOpenDocuments),
    targetDescriptor: nil,
    returnID: AEReturnID(kAutoGenerateReturnID),
    transactionID: AETransactionID(kAnyTransactionID)
)

// Set up the direct object parameter to be a single string holding the
// path to our script.

let parameters = NSAppleEventDescriptor(string: url.path)
event.setDescriptor(parameters, forKeyword: AEKeyword(keyDirectObject))

// The `as NSAppleEventDescriptor?` is required due to a bug in the
// nullability annotation on this method's result (r. 38702068).

var error: NSDictionary? = nil
guard let result = script.executeAppleEvent(event, error: &error) as NSAppleEventDescriptor? else {
    let code = (error?[NSAppleScript.errorNumber] as? Int) ?? 1
    let message = (error?[NSAppleScript.errorMessage] as? String) ?? "…"
    throw NSError(domain: "ShellScript", code: code, userInfo: nil)
}
let scriptResult = result.stringValue ?? ""
```

Service Management

Reply

Posted 3 months ago by eskimo

Add a Comment