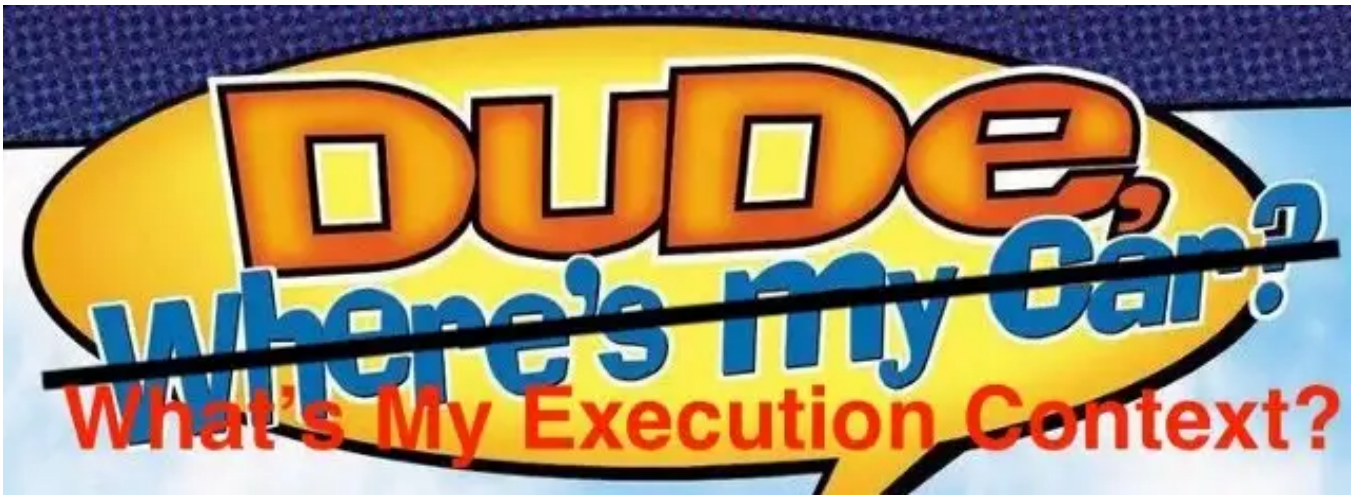
Give Me Some (macOS) Context…

Cedric Owens

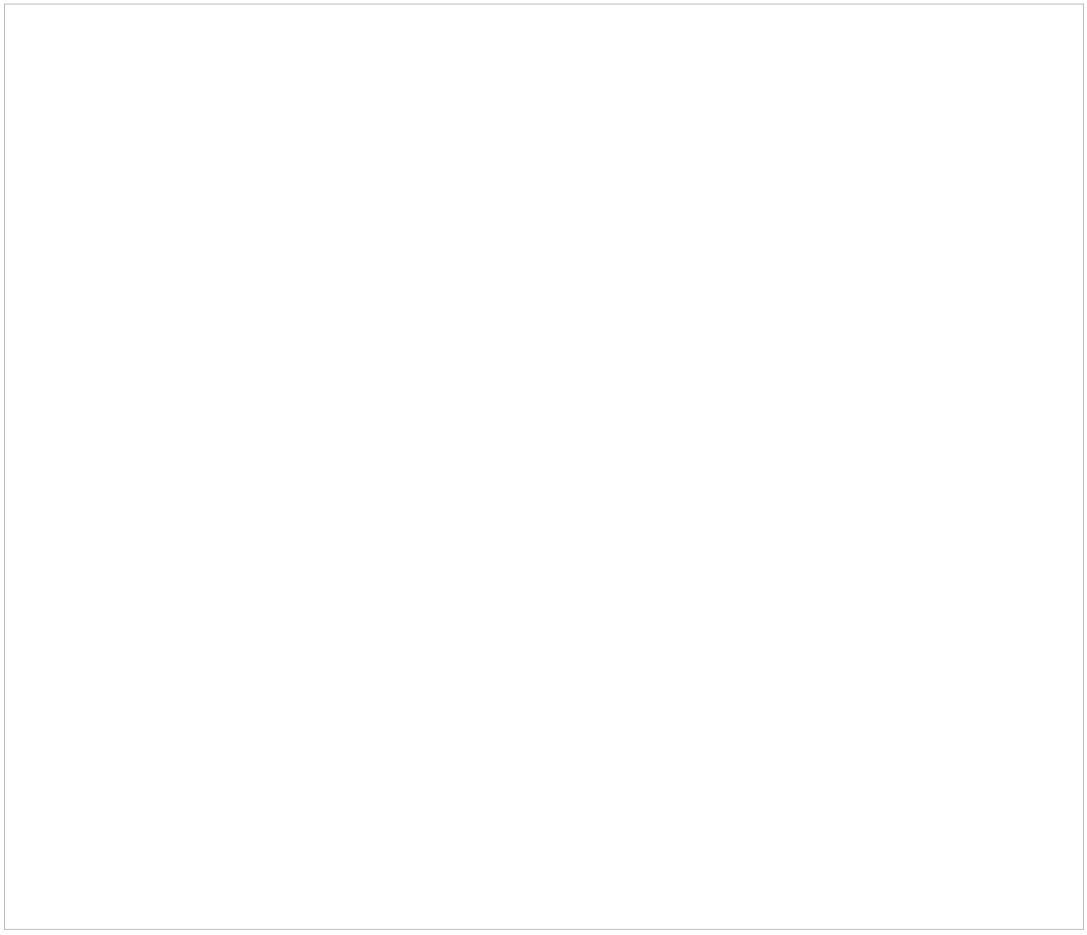


This blog post will dive into what I like to call “execution contexts” on macOS and why it is important to understand these different contexts from a red team perspective when operating on macOS endpoints. I will take a quick look at common payloads on macOS and the execution context of each.

Why?

Before discussing different execution contexts, let’s quickly talk about a key security control specific to macOS. For those who may not be as familiar with macOS, there is a control known as **TCC** (Transparency, Consent, and Control) which was introduced to give macOS users more control over the programs that can access certain portions of macOS that potentially contain sensitive data (ex: ~/Desktop, ~/Documents, ~/Downloads, photos, contacts, etc.). This helps to add additional security barriers so that if an attacker gains remote access to a macOS device, the rogue program will not easily be able to access all parts of macOS without the user explicitly granting access to that rogue program. In this example, the rogue application would generate a TCC prompt to the user when

attempting to access any TCC-protected portion of the operating system. An example TCC prompt:



Sample TCC Prompt

Continuing with this example, it is possible that Terminal has already been granted access to the user's Documents folder (~/Documents). However, since the "TestRogueApp" app bundle is **NOT** running under Terminal's "execution context", it does **NOT** inherit these TCC permissions.

Next, we are going to dig more into how you can identify what "execution context" a program is in on macOS. This can be insightful because there are some instances where the same program can have different execution contexts depending on how it was launched on macOS.

Tell Me More!

Environment variables are super helpful with understanding what your

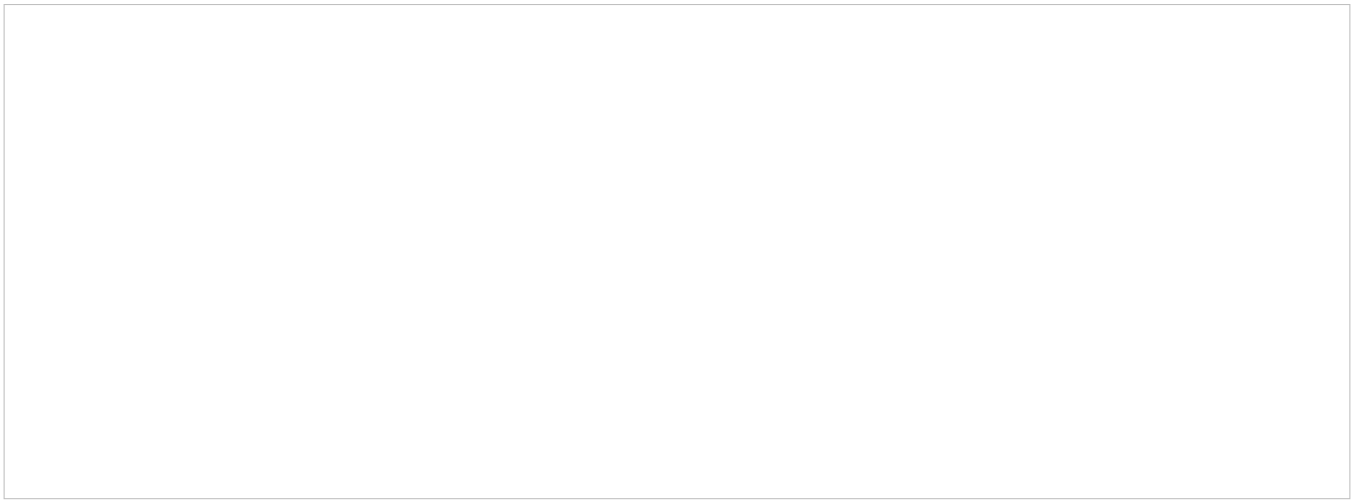
execution context is on macOS. Let's take a look at some examples:

1. **TERMINAL:** If you open a new Terminal window and type `env` you can see some interesting environment variables:

__CFBundleIdentifer Environment Variable Value For Terminal

TERM_PROGRAM Environment Variable Value For Terminal

These two environment variables identify that we are in the execution context of Terminal. This means that we inherit whatever TCC permissions have been granted to Terminal.app. Pretty straightforward. The nice thing about the Terminal context (unlike some other examples below) is that it is native on macOS and there is a good chance that tech savvy users use Terminal and have already granted it some TCC permissions on macOS. Shell scripts and running JXA files via the on-disk osascript binary (ex: osascript [file].js) are a common example of payloads that run under this execution context. Any TCC prompts generated under this execution context would look like:



Sample TCC Prompt Generated When In Terminal's Execution Context

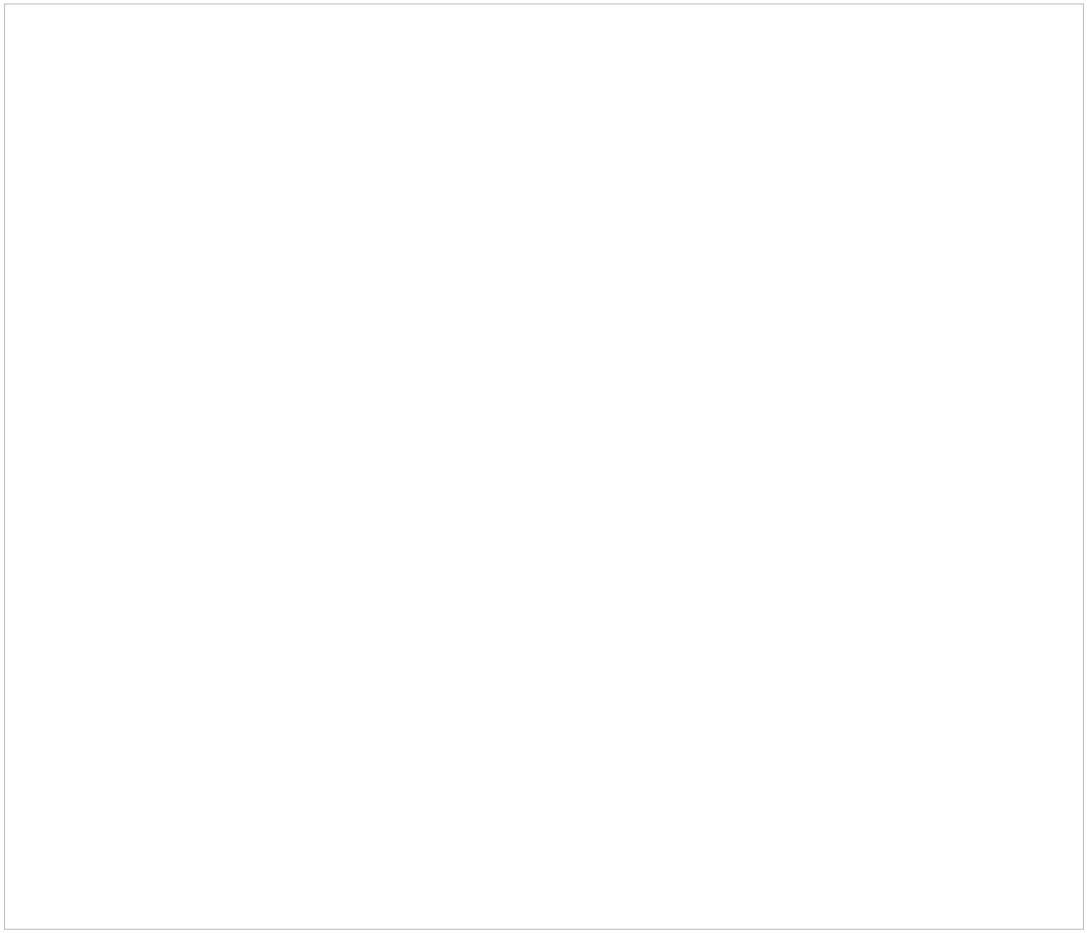
2. **APP BUNDLE:** Going back to my rogue app example, I added code for

the application bundle to print environment variables. Below we can see some interesting environment variables from the rogue app bundle's execution context:

__CFBundleIdentifier Environment Variable For My Test App Bundle

XPC_SERVICE_NAME Environment Variable for My Test App Bundle

Notice that the **__CFBundleIdentifier** is different and instead of having Terminal's bundle ID it now has the bundle ID of the running application. Also notice an additional environment variable that was not present in Terminal's environment variable list: "**XPC_SERVICE_NAME**". These environment variable values let us know that we are in the execution context of an application bundle and therefore we do **NOT** inherit any of Terminal's TCC permission grants from within this context. However if a rogue program can get into the execution context of another application that has certain TCC permissions then that could result in that rogue application gaining unauthorized access (bypassing TCC controls). Any normal request from this app bundle to access any TCC protected folders will generate a pop up such as below:



Sample TCC Prompt from My Rogue App Bundle

3. **INSTALLER PACKAGE:** Let's add a third payload example: an installer package. I created a basic installer package and had it dump its environment variables to a file during execution. Let's take a look at interesting environment variables from within this execution context:

INSTALL_PKG_SESSION_ID Environment Variable For My Test Installer Package

PACKAGE_PATH Environment Variable For My Test Installer Package

My installer package did NOT have the **__CFBundleIdentifer** environment variable (instead it had **INSTALL_PKG_SESSION_ID**). I also noticed **PACKAGE_PATH** which identifies that this execution context is from an Installer Package. This execution context will **NOT** inherit any TCC permissions that Terminal currently has (i.e., any request from this installer

package to access any TCC protected folders will generate a pop up such as below):

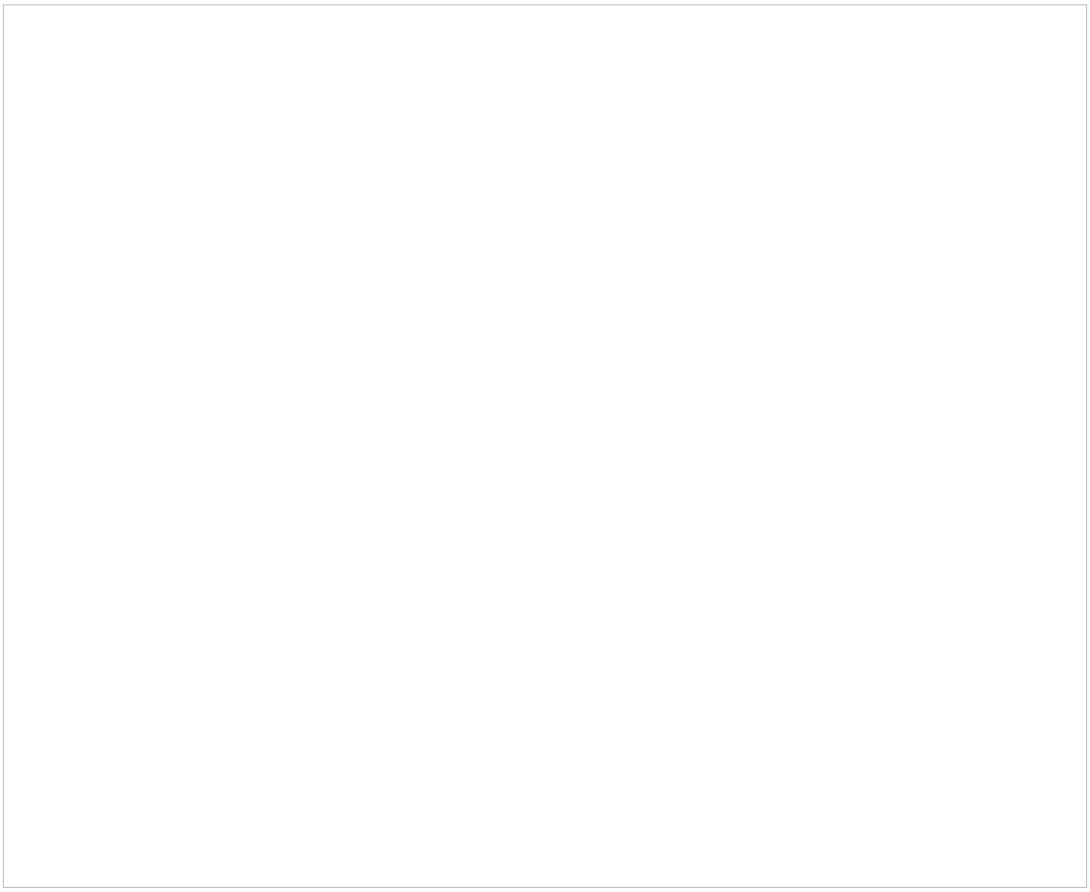
Sample TCC Prompt From My Installer Package

4. For example #4, let's look at a macho binary executed as a launch item (in our example we will use a Launch Agent). I generated a macho binary with code to dump its environment variables, created a simple plist in ~/Library/LaunchAgents to launch this macho binary at startup, and loaded it with `launchctl load`. Here is the interesting environment variable I noticed when in this execution context:

XPC_SERVICE_NAME Environment Variable For macho Binary Run Via Launch Agent

Again, no **__CFBundleIdentifier** environment variable, but instead we do have **XPC_SERVICE_NAME** which contains an identifier of the binary that was run at startup. This execution context will **NOT** inherit any TCC

permissions that Terminal currently has (i.e., any request from this macho binary running as a launch item to access any TCC protected folders will generate a pop up such as below):



Sample TCC Prompt From My macho Binary Running Via Launch Agent Execution

5. .**zshrc EXECUTION:** In our final example we will take this same macho binary that we previously ran as a Launch Agent in #4 (mymacho) and show a simple way that this macho binary can be run under Terminal's context. First, I created a .zshrc file in my home dir (~) and added this to the .zshrc file:

```
nohup /tmp/mymacho &
```

This will execute my macho binary with nohup (i.e., don't stop it) and backgrounded anytime a new Terminal window is opened. I then saved the .zshrc file. Next I opened a new Terminal window and observed the environment variables output by mymacho. Notice the difference this time:

__CFBundleIdentifer Environment Variable When mymacho Is Run By .zshrc

TERM_PROGRAM Environment Variable When mymacho Is Run By .zshrc

Though we are running the same macho binary as before, we are now in Terminal’s context, due to .zshrc execution. Now this context **DOES** inherit Terminal’s TCC Permissions.