Persistent Credential Theft with Authorization Plugins

Archived: 2026-04-06 02:01:32 UTC

Credential theft is often one of the first tactics leveraged by attackers once they've escalated privileges on a victim's machine. Credential theft on OSX has become more difficult with the introduction of System Integrity Protection (SIP). Attackers can no longer use methods such as extracting the master keys from the *securityd* process and decrypting the victim's login keychain. An example of this can be seen here. SIP prevents any user (even root) from modifying system files, directories, and processes. This might lead an attacker to use other methods for credential theft including keylogging, prompting the user for credentials, or triaging the local file system for plaintext credentials. However, there is another alternative called authorization plugins.

Authorization plugins are used to extend the authorization services API and implement mechanisms that are not natively supported by the OS. A legitimate use case would be implementing multi-factor authentication with 3rd party software such as Duo. Each plugin can have several mechanisms, or functions contained within the plugin that implement custom logic to perform authorization. These mechanisms are also present in the policy database. This database contains definitions for various authorization rights. The definition for a particular authorization right can reference several mechanisms, each paired with their corresponding authorization plugin. Apple briefly describes how this works during the logon process here. Essentially, the *loginwindow* process tries to obtain the system.login.console right. The definition in the policy database lists several mechanisms that are executed when this happens. A custom plugin, along with its mechanism, can be inserted into this list. The position of the entry in this list will determine when it will be executed and should be determined based on how the plugin will interact with the OS during the logon process. Additionally, a plugin can be configured to run in a privileged or un-privileged context. This is indicated by the entry string within the policy database. For privileged execution, ",privileged" will need to be appended to the mechanism entry. An author may choose to use an unprivileged plugin if they wish to access the UI and present a prompt to the user. Additional information on plugin context issues can be found here.

An interesting (especially for attackers) feature of authorization plugins is the ability to pass context values between plugins. These values contain important details about the user, including, the home directory, uid, and even the password in clear-text. This feature exists so that developers who wish to utilize both a privileged and un-privileged plugin, can share data between the two in a relatively easy way. Authorization plugins are an ideal credential theft and persistence mechanism for attackers. Persistent credential theft with the ability to execute code at any point during the login process is invaluable. Installing an authorization plugin is a relatively easy process, which we'll cover in the next section.

Installation

Authorization plugins are comprised of two components; the bundle file that contains all of the plugins code, on disk, and an entry into the authorization database. The authorization database is used by the security server to authorize specific rights for a given user, based on rules/policies contained within the database. Applications may change the rules at any given time using the *AuthorizationRightSet*, *AuthorizationRightGet*, and

AuthorizationRightRemove API calls. When adding an entry to the database, the *AuthorizationRightSet* function should be called with a reference to a NSDictionary containing keys and values for each authentication mechanism. An example is shown in the Apple documentation here. Alternatively, the *AuthorizationRightGet* function can be used to obtain the NSDictionary value for an existing right. Then modify the dictionary and add an entry for the custom authorization plugin using the *AuthorizationRightSet* function. A python example of this can be seen here.

The authorization database can also be modified via the security command line tool. If an attacker wanted to add a malicious plugin to the *system.login.console* policy, they would need to first read the policy from the authorization database into a plist file:

```
security authorizationdb read system.login.console >> system.login.console.plist
```

Then insert their bundle as one of the mechanisms in the following format:

```
<string>[Plugin Name]:auth,privileged</string>
```

PlistBuddy can also do this with the following command:

```
/usr/libexec/PlistBuddy -c “add :mechanisms:[HomeDir Offset] string [bundlename]:login,privileged” <Plist file
```

Once the plist file has been updated, sync this change to the database with the following command:

```
security authorizationdb write system.login.console < /path/to/plist/file
```

Weaponization

Let’s walk through creating a plugin with Xcode and then add in logic for payload execution. In this example we’ll use a shell command to launch an Apfell payload, but a python empire payload will work as well.

1. First open Xcode and choose the “create a new xcode project option”.
2. Under macOS, select the bundle template. Give the project a name and then click next.
    Xcode Bundle Project Template
3. Next, download and save all three files listed here in the project folder. Add all three files to the project by right-clicking the project folder in the navigation pane and then select add files to “[your project name]”.
   Adding files to the project
4. Then, setup Apfell and generate a JXA payload. Refer to this blog post by @its_a_feature_ for setup and payload creation instructions.
5. Copy the JXA one-liner that begins with *osascript -l JavaScript* and drop it into the appropriate location.
    Adding the JXA payload to the project

6. Select product from the top menu bar and then select build. The resulting bundle will be shown under the product directory on the left navigation pane. Right-click on the bundle file and then select show in finder. Copy the file to an accessible location of your choosing. Building the project

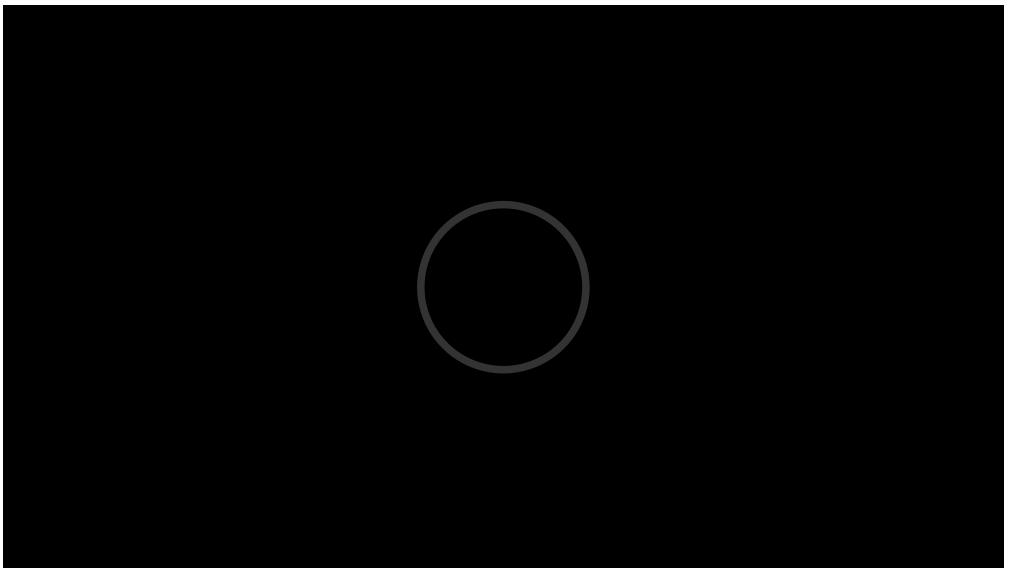
 Show the bundle's location in finder

Now the plugin should be ready for installation. Use the command line tool or the Authorization APIs described in the installation section to accomplish this.

Once the plugin is installed, either restart or complete a logon/logoff cycle to trigger execution. Note that the plugin will be loaded into the authorizationhost process located here:

```
/System/Library/Frameworks/Security.framework/Versions/A/MachServices/authorizationhost.bundle/Contents/MacOS/a
```

Now for a quick demo of installing and executing an authorization plugin. You can find the Xcode project used for this demo here.



Detection

For detection considerations, a solid starting point would be to create rules based on file creation in the /Library/Security/SecurityAgentPlugins directory. Authorization plugins can only be installed in that directory. An example osquery rule for file creation events should look like this:

```
SELECT * FROM file_events WHERE target_path LIKE '/Library/Security/SecurityAgentPlugins/%' AND action = 'adde
```

Command line signatures based on usage of the security binary to modify the authorization database should be moderately effective for detection. Additionally, for non-enterprise user's, I would highly recommend installing some of the security software offered by Objective-See. *KnockKnock*, the persistence scanner, has had detection rules for authorization plugins since 2015!

Authorization plugins are a high risk/reward method of persistence and credential theft. If something goes a wry with the the plugins' installation, the victim user will almost certainly be suspicious and may alert defenders to your presence. On the flipside, they provide an effective method for credential theft and code execution. Attackers can continually harvest credentials even in the event that the end-user changes their password. Similarly, you can accomplish this on windows with mimikatz and it's Security Support Provider.
