Debugging a Network Extension Provider

This thread has been locked by a moderator.

328

I regularly see folks struggle to debug their Network Extension providers. For an app, and indeed various app extensions, debugging is as simple as choosing Product > Run in Xcode. That's not the case with a Network Extension provider, so I thought I'd collect together some hints and tips to help you get started.

If you have any comments or questions, create a new thread here on DevForums and tag it with *Network Extension* so that I see it.

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

Debugging a Network Extension Provider

Debugging a Network Extension provider presents some challenges; its not as simple as choosing Product > Run in Xcode. Rather, you have to run the extension first and then choose Debug > Attach to Process. Attaching is simple, it's the running part that causes all the problems. When you first start out it can be a challenge to get your extension to run at all.

Add a First Light Log Point

The first step is to check whether the system is actually starting your extension. My advice is to add a first light log point, a log point on the first line of code that you control. The exact mechanics of this depend on your development, your deployment target, and your NE provider's packaging. In all cases, however, I recommend that you log to the system log.

The system log has a bunch of cool features. If you're curious, see Your Friend the System Log. The key advantage is that your log entries are mixed in with system log entries, which makes it easier to see what else is going on when your extension loads, or fails to load.

IMPORTANT Use a unique subsystem and category for your log entries. This makes it easier to find them in the system log.

For more information about Network Extension packaging options, see TN3134 Network Extension provider deployment.

Logging in Swift

If you're using Swift, the best logging API depends on your deployment target. On modern systems — macOS 11 and later, iOS 14 and later, and aligned OS releases — it's best to use the `Logger` API, which is shiny and new and super Swift friendly. For example:

```
let log = Logger(subsystem: "com.example.galactic-mega-builds", category: "earth")

let client = "The Mice"
let answer = 42
log.log(level: .debug, "run complete, client: \(client), answer: \(answer, privacy: .private)")
```

If you support older systems, use the older, more C-like API:

```
let log = OSLog(subsystem: "com.example.galactic-mega-builds", category: "earth")

let client = "The Mice"
let answer = 42
os_log(.debug, log: log, "run complete, client: %@, answer: %{private}d", client as NSString, answer)
```

Logging in C

If you prefer a C-based language, life is simpler because you only have one choice:

```
#import <os/log.h>

os_log_t log = os_log_create("com.example.galactic-mega-builds", "earth");

const char * client = "The Mice";
int answer = 42;
os_log_debug(log, "run complete, client: %s, answer: %{private}d", client, answer);
```

Add a First Light Log Point to Your App Extension

If your Network Extension provider is packaged as an app extension, the best place for your first light log point is an override of the provider's initialiser. There are a variety of ways you could structure this but here's one possibility:

```
import NetworkExtension
import os.log

class PacketTunnelProvider: NEPacketTunnelProvider {

    static let log = Logger(subsystem: "com.example.myvpnapp", category: "packet-tunnel")

    override init() {
        self.log = Self.log
        log.log(level: .debug, "first light")
        super.init()
    }

    let log: Logger

    … rest of your code here …
}
```

This uses a Swift static property to ensure that the log is constructed in a race-free manner, something that's handy for all sorts of reasons.

It's possible for your code to run before this initialiser — for example, if you have a C++ static constructor — but that's something that's best to avoid.

Add a First Light Log Point to Your System Extension

If your Network Extension provider is packaged as a system extension, add your first light log point to `main.swift`. Here's one way you might structure that:

```
import NetworkExtension

func main() -> Never {
    autoreleasepool {
        let log = PacketTunnelProvider.log
        log.log(level: .debug, "first light")
        NEProvider.startSystemExtensionMode()
    }
    dispatchMain()
}

main()
```

See how the `main` function gets the `log` object from the static property on `PacketTunnelProvider`. I told you that'd come in handy (-:

Again, it's possible for your code to run before this but, again, that's something that's best to avoid.

App Extension Hints

Both iOS and macOS allow you to package your Network Extension provider as an app extension. On iOS this is *super* reliable. I've never seen any weirdness there.

That's not true on macOS. macOS lets the user put apps anywhere; they don't have to be placed in the Applications directory. macOS maintains a database, the Launch Services database, of all the apps it knows about and their capabilities. The app extension infrastructure uses that database to find and load app extensions. It's not uncommon for this database to get confused, which prevents Network Extension from loading your provider's app extension. This is particularly common on developer machines, where you are building and rebuilding your app over and over again.

The best way to avoid problems is to have a single copy of your app extension's container app on the system. So, while you're developing your app extension, delete any other copies of your app that might be lying around.

If you run into problems you may be able to fix them using:

- `lsregister`, to interrogate and manipulate the Launch Services database
- `pluginkit`, to interrogate and manipulate the app extension state [1]

IMPORTANT Both of these tools are for debugging only; they are not considered API. Also, `lsregister` is not on the default path; find it at `/System/Library/Frameworks/CoreServices.framework/Frameworks/LaunchServices.framework/Versions/A/Support/lsregister`.

For more details about `pluginkit`, see the `pluginkit` man page.

When debugging a Network Extension provider, add buttons to make it easy to save and remove your provider's configuration. For example, if you're working on a packet tunnel provider you might add:

- A Save Config button that calls the `saveToPreferences(completionHandler:)` method to save the tunnel configuration you want to test with
- A Remove Config button that calls the `removeFromPreferences(completionHandler:)` method to remove your tunnel configuration

These come in handy when you want to start again from scratch. Just click Remove Config and then Save Config and you've wiped the slate clean.

You don't have to leave these buttons in your final product, but it's good to have them during bring up.

[1] This tool is named after the PluginKit framework, a private framework used to load this type of app extension. It's distinct from the ExtensionKit framework which is a new, public API for managing extensions.

Sytem Extension Hints

macOS allows you to package your Network Extension provider as a system extension. For this to work the container app must be in the Applications directory [1]. Copying it across each time you rebuild your app is a chore. To avoid that, add a Build post-action script:

1. Select your app's scheme and choose Product > Scheme > Edit Scheme.
2. On the left, select Build.
3. Click the chevron to disclose all the options.
4. Select Post-actions.
5. In the main area, click the add (+) button and select New Run Script Action.
6. In the "Provide build settings from" popup, select your app target.
7. In the script field, enter this script:

```
ditto "${BUILT_PRODUCTS_DIR}/${FULL_PRODUCT_NAME}" "/Applications/${FULL_PRODUCT_NAME}"
```

Now, each time you build your app, this script will copy it to the Applications directory. Build your app now, both to confirm that this works and to enable the next step.

The next issue you'll find is that choosing Product > Run runs the app from the build products directory rather than the Applications directory. To fix that:

1. Edit your app's scheme again.
2. On the left, select Run.
3. In the main area, select the Info tab.
4. From the Executable popup, choose Other.
5. Select the copy of your app in the Applications directory.

Now, when you choose Product > Run, Xcode will run that copy rather than the one in the build products directory. Neat-o!

For your system extension to run your container app must activate it. As with the Save Config and Remote Config buttons described earlier, it's good to add easy-to-access buttons to activate and deactivate your system extension.

With an app extension the system automatically terminates your extension process when you rebuild it. This is not the case with a system extension; you'll have to deactivate and then reactivate it each time. Each activation must be approved in System Settings > Privacy & Security. To make that easier, leave System Settings running all the time.

This debug cycle leaves deactivated but not removed system extensions installed on your system. These go away when you restart, so do that from time to time. Once a day is just fine.

macOS includes a tool, `systemextensionsctl`, to interrogate and manipulate system extension state. The workflow described above does not require that you use it, but it's good to keep in mind. Its man page is largely content free so run the tool with no arguments to get help.

[1] Unless you disable System Integrity Protection, but who wants to do that?

You Can Attach with the Debugger

Once your extension is running, attach with the debugger using one of two commands:

- To attach to an app extension, choose Debug > Attach to Process > *YourAppExName*.
- To attach to a system extension, choose Debug > Attach to Process by PID or Name. Make sure to select Debug Process As root. System extensions run as root so the attach will fail if you select Debug Process As Me.

But Should You?

Debugging networking code with a debugger is less than ideal because it's common for in-progress network requests to time out while you're stopped in the debugger. Debugging Network Extension providers this way is especially tricky because of the extra steps you have to take to get your provider running. So, while you can attach with the debugger, and that's a great option in some cases, it's often better not to do that.

Rather, consider the following approach:

- Write the core logic of your provider so that you can unit test each subsystem outside of the provider. This may require some scaffolding but the time you take to set that up will pay off once you encounter your first gnarly problem.
- Add good logging to your provider to help debug problems that show up during integration testing.

I recommend that you treat your logging as a feature of your product. Carefully consider where to add log points and at what level to log. Check this logging code into your source code repository and ship it — or at least the bulk of it — as part of your final product. This logging will be super helpful when it comes to debugging problems that only show up in the field.

Remember that, when using the system log, log points that are present but don't actually log anything are very cheap. In most cases it's fine to leave these in your final product.

Now go back and read Your Friend the System Log because it's full of useful hints and tips on how to use the system log to debug the really hard problems.

General Hints and Tips

Install the Network Diagnostics and VPN (Network Extension) profiles [1] on your test device. These enable more logging and, most critically, the recording of private data. For more info about that last point, see… you guessed it… Your Friend the System Log.

Get these profiles from our Bug Reporting > Profiles and Logs page.

When you're bringing up a Network Extension provider, do your initial testing with a tiny test app. I regularly see folks start out by running Safari and that's less than ideal. Safari is a huge app with lots of complexity, so if things go wrong it's hard to tell where to look.

I usually create a small test app to use during bring up. The exact function of this test app varies by provider type. For example:

- If I'm building a packet tunnel provider, I might have a test function that makes an outgoing TCP connection to an IP address. Once I get that working I add another function that makes an outgoing TCP connection to a DNS name. Then I start testing UDP. And so on.
- Similarly for a content filter, but then it makes sense to add a test that runs a request using `URLSession` and another one to bring up `WKWebView`.
- If I'm building a DNS proxy provider, my test app might use `CFHost` to run a simple name-to-address query.

Also, consider doing your bring up on the Mac even if your final target is iOS. macOS has a bunch of handy tools for debugging networking issues, including:

- `dig` for DNS queries
- `nc` for TCP and UDP connections
- `netstat` to display the state of the networking stack
- `tcpdump` for recording a packet trace [2]

Read their respective man pages for all the details.

[1] The latter is not a profile on macOS, but just a set of instructions.

[2] You can use an RVI packet trace on iOS but it's an extra setup step.

Revision History

- **2023-04-02** Fixed one of the steps in *System Extension Hints*.

Network Extension

Reply

Posted 2 months ago by eskimo

Add a Comment