Using a Sysdiagnose Log to Debug a Hard-to-Reproduce Problem

This thread has been locked by a moderator.

12

I regularly talk to developers debugging hard-to-reproduce problems. I have some general advice on that topic. I've posted this to DevForums before, and also sent similar info to folks who've opened a DTS incident, but I figured I should write it down properly.

If you have questions or comments, put them in a new thread here on DevForums. Tag it with *Debugging* so that I see it.

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

Using a Sysdiagnose Log to Debug a Hard-to-Reproduce Problem

Some problems are hard to reproduce in your office. These usually fall into one of two categories:

- Environment specific — This is where some of your users can easily reproduce the problem, but you can't reproduce it in your environment.
- Intermittent — In this case the problem could affect any user, but it's hard to predict when a given user will see the problem.

A key tool in debugging such problems is the sysdiagnose log. This post explains how to make this technology work for you.

IMPORTANT A sysdiagnose log might contain private information. If you ask a user to send you a log, make sure they understand the privacy impact of that. If you want to see how Apple handles this, run the `sysdiagnose` command on a fresh Mac and read through it's initial prompt.

Sysdiagnose Logs

All Apple platforms can generate sysdiagnose logs. For instructions on how to do this, see our Bug Reporting > Profiles and Logs page.

The resulting log is a `.tar.gz` file. Unpacking that reveals a bunch of files. The most critical of these is `system_logs.logarchive`, which is a snapshot of the system log. For more information about the system log, including links to the documentation, see Your Friend the System Log.

This log snapshot includes many thousands of log entries (I just took a log snapshot on my Mac and it had 22.8 *million* log entries!). That can be rather daunting. To avoid chasing your tail, it pays to do some preparation.

Preparation

The goal here is to create a set of instructions that you can give to your user to capture an actionable sysdiagnose log. That takes some preparation.

To help orient yourself in the log, add log points to your code to highlight the problem. For example, if you're trying to track down a keychain problem where `SecItemCopyMatching` intermittently fails with `errSecMissingEntitlement` (-34018), add a log point like this:

```
import os.log

let log = Logger(subsystem: "com.example.waffle-varnish", category: "keychain")

func … {
    let err = SecItemCopyMatching(…)
    log.log("SecItemCopyMatching failed, err: \(err)")
}
```

When you look through a log, find this specific failure by searching for `SecItemCopyMatching failed, err: -34018`.

You might also add log points at the start and end of an operation, which helps establish a time range of interest.

Log points like this have a very low overhead and it's fine to leave them enabled in your released product. However, in some cases you might want to make more extensive changes, creating a debug build specifically to help investigate your problem. Think about how you're going to get that debug build to the affected users. You might, for example, set up a special TestFlight group for folks who've encountered this issue.

Go to Bug Reporting > Profiles and Logs and look for debug profiles that might help your investigation. For example, if you're investigating a Network Extension issue, the *VPN (Network Extension)* debug profile will enable useful debug logging.

Now craft your instructions for your user. Include things like:

- Your take on the privacy impact on this
- Instructions on how to get the necessary build of your product
- If there's a debug profile, instructions on how to install that
- Instructions on how to trigger the sysdiagnose log
- And on how to send it to you

IMPORTANT Make sure to stress how important it is that the user triggers the sysdiagnose immediately after seeing the problem.

Finally, test your steps. Do an initial test in your office, to make sure that the log captures the info you need. Then do an end-to-end test with someone who's about as technically savvy as your users, to make sure that your instructions make sense to Real People™.

Prompting for a Sysdiagnose Log

In some cases it might not be obvious to the user when to trigger a sysdiagnose log. Imagine you're hunting the above-mentioned `errSecMissingEntitlement` error and it only crops up when your product is performing some task in the background. The user doesn't see that failure, they're not even running your app!, so they don't know that action is required.

A good option here is to add code to actively monitor for the failure and post a local notification requesting that the user trigger a sysdiagnose log. Continuing the above example, you might write code like this:

```
func … {
    let err = SecItemCopyMatching(…)
    log.log("SecItemCopyMatching failed, err: \(err)")
    if err == errSecMissingEntitlement {
        … post a local notification …
    }
}
```

Obviously this is quite intrusive so, depending on the market for your product, you might not want to deploy this to all users. Perhaps you can restrict it to your internal testers, or your external beta testers, or a particularly savvy set of customers.

Looking at the System Log

Once you have your sysdiagnose log, unpack it and open the system log snapshot (`system_logs.logarchive`) in Console. The hardest part is knowing where to start. That's why adding your own log entries, as discussed in *Preparation*, is so important. A good general process is:

1. Search for log entries from your subsystem. An easy way to initiate that search is to paste the text `subsystem:SSS`, where `SSS` is your subsystem, into the Search field. Continuing the above example, find that log entry by pasting in `subsystem:com.example.waffle-varnish`.
2. Identify the log entry that indicates the problem and select it.
3. Then remove your search and work backwards through the log looking for system log entries related to your issue.

The relevant log entries might not be within the time range shown by Console. Customise that by selecting values from the Showing popup in the pane divider. Once you have a rough idea of the timeframe involved, select Custom from that popup to focus on that range.

If the log is showing stuff that's not relevant to your problem, Console has some great facilities for filtering those out. For the details, choose Help > Console Help.

Talk to Apple

A key benefit of this approach is that, if your investigation suggests that this is a system bug, you can file a bug report and attach this sysdiagnose log to it. The setup described above is exactly the sort of info needed to analyse the bug.

Likewise, if you decide to seek formal code-level support by opening a DTS tech support incident, your friendly neighbourhood DTS engineer will find that sysdiagnose log very handy.

Debugging OSLog

Reply

Posted 4 days ago by eskimo

Add a Comment