Sample Code

Communicating between a DriverKit extension and a client app

Send and receive different kinds of data securely by validating inputs and asynchronously by storing and using a callback.

Download

DriverKit 20.0+ | iOS 17.0+ | iPadOS 17.0+ | macOS 14.0+ | Xcode 15.0+

Overview

This sample code project shows how a DriverKit extension (dext) receives data from an app client running on macOS or iPadOS. The sample handles both scalar data and structures, and has two code paths for each type: an insecure version, and a “checked” version that validates traits like data size and input count.

The sample also demonstrates registering and executing a callback function, so the driver can call the client asynchronously.

> Note
>
> This version updates the sample to run in iPadOS. DriverKit on iPadOS requires an iPad with an M-series chip.

The sample project contains two targets:

- `DriverKitSampleApp` - A SwiftUI app for macOS and iPadOS. Use this app to install or update the driver, and then to call the driver directly.
- `NullDriver` - The dext itself, which responds to client calls and optionally checks that each call sends the expected data.

Configure the Sample Code Project

To run the sample code project, you first need to build and run `DriverKitSampleApp`, which installs the driver.

You can set up the project to build with automatic signing (recommended) or manual signing. To use automatic signing:

1. Temporarily turn off SIP, as described in the article Disabling and Enabling System Integrity Protection. After you've done this, confirm that SIP is disabled with the Terminal command `csrutil status`, and enter dext development mode with `systemextensionsctl developer`, as described in the article Debugging and Testing System Extensions.

2. Select the `DriverKitUserClientSample` project and use the "Signing & Capabilities" tab to set the `DriverKitSampleApp` and `NullDriver` targets to "Automatically manage signing".

3. In the "Build Settings" tab, change the "Code Signing Identity" value to "Sign to Run Locally" for both targets.

4. Also in "Build Settings", add a new build setting with the name `AD_HOC_CODE_SIGNING_ALLOWED` and the value `YES` for Debug schemes. This build setting is already present in the downloadable version of the project; you need to add it to your own projects to build and run them automatically.

If you want to build with manual signing instead, do the following:

1. Choose new bundle identifiers for the app and driver. The bundle identifiers included with the project already have App IDs associated with them, so you need unique identifiers to create your own App IDs. Use a reverse-DNS format for your identifier, as described in Preparing Your App For Distribution.

2. In the Xcode Project navigator, choose the project and use the Signing & Capabilities tab to replace the existing bundle identfier with your chosen identifier for each of the targets.

3. Request entitlements, as described in Requesting Entitlements for DriverKit Development. For `DriverKitSampleApp`, you need the `com.apple.developer.system-extension.install` and Communicates with Drivers entitlements. For `NullDriver`, you need the `com.apple.developer.driverkit` entitlement and the `com.apple.developer.driverkit.allow-any-userclient-access` entitlements. This latter macOS-only entitlement allows any app to connect to the driver as a user client. Although this simplifies running the sample code, in your own apps you may prefer to use `com.apple.developer.driverkit.userclient-access`. This entitlement goes on the app rather than the driver, and lists bundle identifiers of drivers it can connect to.

4. On `developer.apple.com`, select Account and visit the "Certificates, Identifiers, and Profiles" section. Select "Identifiers" and create new App IDs for `DriverKitSampleApp` and `NullDriver`. For the Bundle ID, choose "explicit", and use the names you chose in the first step. When you reach the "Capabilities" step, `DriverKitSampleApp` needs the

"Communicates with Drivers" and "System Extension" capabilities. For `NullDriver`, under the "Capabilities" tab, enable the "DriverKit (development)" capability, and under the "Additional Capabilities" tab, enable "DriverKit" and "DriverKit Allow Any UserClient Access".

5. For each of the App IDs you created in the previous steps, select Profiles to create a new provisioning profile. You need one for the macOS and iOS host app, and one for the driver. When creating the driver's profile, be sure to select DriverKit App Development as the profile type.

6. Download each profile and add it to Xcode.

7. In the "Signing & Capabilities" tab, set each target to manual code signing and select the newly-created profile.

Install and run the driver extension on macOS

To run the sample app in macOS, use the scheme selector to select the `DriverKitSampleApp` scheme and the My Mac destination. Build the target, then copy the app to the Applications folder and launch the app. For simplicity, you can run the app directly from Xcode instead, without moving the app bundle to the Applications folder each time. To do this, use the `systemextensionsctl` command to enable system extensions developer mode, as explained in Debugging and testing system extensions.

The `DriverKitSampleApp` target declares the `NullDriver` as a dependency, so building the app target builds the dext and its installer together. When run, the `DriverKitSampleApp` shows a single window with a text label that says "NullDriver isn't loaded". Below this, it shows an "Install Dext" button, and a "Communicate with Dext" button. Click "Install Dext" to perform the installation.

To install the dext on macOS, the app uses the System Extensions framework to install and activate the dext, as described in Installing System Extensions and Drivers.

```swift
let request = OSSystemExtensionRequest
    .activationRequest(forExtensionWithIdentifier: dextIdentifier,
                       queue: .main)
request.delegate = self
OSSystemExtensionManager.shared.submitRequest(request)
```

Note

This call may prompt a “System Extension Blocked” dialog, which explains that `DriverKit SampleApp` tried to install a new system extension. To complete the installation, open System Preferences and go to the Security & Privacy pane. Unlock the pane if necessary, and click “Allow” to complete the installation. To confirm installation of the `NullDriver` extension, run `systemextensionsctl list` in Terminal.

Run the sample in iPadOS

To run the sample app in iPadOS, connect an iPad with an M-series chip to your Mac. Use the scheme selector to select the `DriverKitSampleApp` scheme and the name of your iPad as the destination. Run the app directly from Xcode to launch it on your iPad.

In iPadOS, the `DriverKitSampleApp` app doesn’t show the Driver Manager section because the app isn’t responsible for installing the driver like it is in macOS. Instead, open the Settings app, navigate to Privacy & Security > Drivers, and enable the driver there. You can also navigate to this settings panel by tapping “Open Settings to Enable Driver” in DriverKitSampleApp.

Call the driver from the client

After installing the driver and granting it permission to run in the Settings app, you can use DriverKitSampleApp to communicate with the driver. Click the “Communicate with Dext” button to connect to the driver.

To find the driver, the host app uses the `dextIdentifier` to create a matching dictionary and registers this with `IOServiceAddMatchingNotification`. When the system finds a running driver with the matching identifier, it calls the app’s `DeviceAdded` method.

```
CFMutableDictionaryRef matchingDictionary = IOServiceNameMatching(dextIdentifier);
if (matchingDictionary == NULL)
{
    fprintf(stderr, "Failed to initialize matchingDictionary.\n");
    UserClientTeardown();
    return false;
}
matchingDictionary = (CFMutableDictionaryRef)CFRetain(matchingDictionary);
matchingDictionary = (CFMutableDictionaryRef)CFRetain(matchingDictionary);

ret = IOServiceAddMatchingNotification(globalNotificationPort, kIOFirstMatchNotifica
if (ret != kIOReturnSuccess)
```

```
{
    fprintf(stderr, "Add matching notification failed with error: 0x%08x.\n", ret);
    UserClientTeardown();
    return false;
}
DeviceAdded(refcon, globalDeviceAddedIter);
```

When DeviceAdded runs, it iterates over matching services until it finds one that it can connect to with `IOServiceOpen`. If the IOServiceOpen call succeeds, the app calls an internal Swift DeviceAdded method to update the SwiftUI data model with a connection to the driver.

```
void DeviceAdded(void* refcon, io_iterator_t iterator)
{
    kern_return_t ret = kIOReturnSuccess;
    io_connect_t connection = IO_OBJECT_NULL;
    io_service_t device = IO_OBJECT_NULL;
    bool attemptedToMatchDevice = false;

    while ((device = IOIteratorNext(iterator)) != IO_OBJECT_NULL)
    {
        attemptedToMatchDevice = true;

        // Open a connection to this user client as a server to that client, and sto
        ret = IOServiceOpen(device, mach_task_self_, 0, &connection);

        if (ret == kIOReturnSuccess)
        {
            fprintf(stdout, "Opened connection to dext.\n");
        }
        else
        {
            fprintf(stderr, "Failed opening connection to dext with error: 0x%08x.\
            IOObjectRelease(device);
            return;
        }

        SwiftDeviceAdded(refcon, connection);

        IOObjectRelease(device);
    }
}
```

The SwiftUI “Driver Communication” view shows three sets of buttons, along with a “Manage Dext” button to return to the installer view. The sets of buttons that communicate with the driver are:

- “Unchecked” — “Scalar”, “Struct”, and “Large Struct”
- “Checked” — “Scalar” and “Struct”
- “Async” — “Assign Callback” and “Async Action”

When you click or tap one of these buttons, the “Waiting for action” label changes to “Request returned successfully” if the call to the driver succeeds. If an error occurs, the label says “Request returned an error, check the logs for details”. If you launched the app from Xcode, view the Console to see the data sent to and returned from the driver.

The buttons in the Unchecked and Checked sections exercise different code paths that send scalar values and structures to the dext. Note that these are synchronous calls that block until the driver returns a result. The buttons in the Async section perform asynchronous operations that allow the driver to call back to the client after a delay.

Each of these options uses the connection in calls to \`IOConnectCallScalarMethod\`, \`IOConnectCallStructMethod\`, and \`IOConnectCallAsyncStructMethod\` (or \`IOConnectCallMethod\` and \`IOConnectCallAsyncMethod\`, which this sample doesn’t use). For example, the following listing shows the Scalar call from the Unchecked button group, which sends an array of 16 `uint64_t` values, and receives a different array back.

```
kern_return_t ret = kIOReturnSuccess;

// IOConnectCallScalarMethod will fail intentionally for any inputCount or outputCou
const uint32_t arraySize = 16;
const uint64_t input[arraySize] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1

uint32_t outputArraySize = arraySize;
uint64_t output[arraySize] = {};

ret = IOConnectCallScalarMethod(connection, MessageType_Scalar, input, arraySize, ou
if (ret != kIOReturnSuccess)
{
    printf("IOConnectCallScalarMethod failed with error: 0x%08x.\n", ret);
    PrintErrorDetails(ret);
}
```

The other options are all similar, differing only in which `IOConnect...` function they call and the type of data they send.

Validate arguments to driver function calls

The `NullDriver` receives calls from the client in its overridden `ExternalMethod` method. The Unchecked options in the SwiftUI view perform calls that the driver passes unchecked to its `ExternalMethod` implementation. In practice, it's important that a driver validates its inputs before passing them along, to make sure the data is the expected size and contains reasonable values. `NullDriver` has functions that check scalar and structure calls, which are exercised by the Checked buttons in the SwiftUI app.

The "checked" methods in `NullDriver` — `CheckedScalar` and `CheckedStruct` — use an `IOUserClientMethodDispatch` instance to describe the expected fields of the `IOUserClientMethodArguments`. The sample stores these dispatch instances in an array called `externalMethodChecks`. For example, the dispatch instance for the checked scalar call expects to receive and return 16 scalar values, as seen below:

```
[ExternalMethodType_CheckedScalar] =
{
    .function = (IOUserClientMethodFunction) &NullDriverUserClient::StaticHandleExte
    .checkCompletionExists = false, // Since this call doesn't use a callback, this
    .checkScalarInputCount = 16,
    .checkStructureInputSize = 0,
    .checkScalarOutputCount = 16,
    .checkStructureOutputSize = 0,
},
```

After fetching the appropriate `IOUserClientMethodDispatch` instance from the array, the driver passes it in its call to the superclass's `ExternalMethod` along with the method selector and its arguments. If the number of arguments or return values don't match what's in the dispatch instance, the call fails and returns `kIOReturnBadArgument`. Checking client calls like this prevents a malicious call to the driver from using attack vectors like buffer overruns.

Prepare for driver-to-client callbacks

`DriverKitSampleApp` also shows how to communicate from the driver to the client by using a callback function. The Assign Callback button sets up a callback to make an asynchronous call to the client, and then invokes the callback after a short delay to simulate the driver acting on its own. After registering a callback with this button, calls to Async Action re-invoke the callback.

The `NullDriver` class defines `NullDriver_IVars`, the DriverKit structure that holds the driver's instance variables. `NullDriver_IVars` stores the callback action, as well as a dispatch queue and a timer dispatch source to use when calling back to the client.

```
struct NullDriverUserClient_IVars {
    OSAction* callbackAction = nullptr;
    IODispatchQueue* dispatchQueue = nullptr;
    IOTimerDispatchSource* dispatchSource = nullptr;
    OSAction* simulatedAsyncDeviceResponseAction = nullptr;
};
```

NullDriver initializes the dispatchQueue and dispatchSource in its Start implementation.

The driver's implementation of Start also sets up the ivars member simulatedAsyncDeviceResponseAction, which the example uses to simulate asynchronous processing that happens on real hardware. This OSAction refers to an asynchronous timer callback to the SimulatedAsyncEvent function defined in the .iig file:

```
virtual void SimulatedAsyncEvent(OSAction* action, uint64_t time) TYPE(IOTimerDispat
```

This declaration takes the same arguments as the IOTimerDispatchSource::TimerOccurred method that that the TYPE macro wraps. By declaring the callback's name as SimulatedAsyncEvent, the TYPE macro synthesizes CreateActionSimulatedAsyncEvent, the function that creates the OSAction. The driver's Start implementation then calls this synthesized method to initialize the simulatedAsyncDeviceResponseAction member of the ivars structure:

```
ret = CreateActionSimulatedAsyncEvent(sizeof(DataStruct), &ivars->simulatedAsyncDevi
if (ret != kIOReturnSuccess)
{
    Log("Start() - Failed to create action for simulated async event with error: 0x%
    goto Exit;
}
```

Retain and use the callback to notify the client

When the driver is running and it receives a request from the client to register a callback, it calls NullDriver::RegisterAsyncCallback. This method stores the completion, if it exists, in the ivars structure, like this:

```
if (arguments->completion == nullptr)
{
    Log("Got a null completion.");
    return kIOReturnBadArgument;
}

// Save the completion for later.
// If not saved, then it might be freed before the asychronous return.
ivars->callbackAction = arguments->completion;
ivars->callbackAction->retain();
```

Next, the `NullDriver::RegisterAsyncCallback` method sets up a delayed callback to the client to simulate a hardware delay, allowing it to return quickly, by using the `simulatedAsyncDeviceResponseAction`:

```
input = (DataStruct*)arguments->structureInput->getBytesNoCopy();

// Retain action memory for later work.
void* osActionRetainedMemory = ivars->simulatedAsyncDeviceResponseAction->GetReferer
memcpy(osActionRetainedMemory, input, sizeof(DataStruct));

output.foo = input->foo + 1;
output.bar = input->bar + 10;

arguments->structureOutput = OSData::withBytes(&output, sizeof(DataStruct));

// Dispatch action that waits five to seven seconds and then calls the callback.
const uint64_t fiveSecondsInNanoSeconds = 5000000000;
const uint64_t twoSecondsInNanoSeconds = 2000000000;
uint64_t currentTime = clock_gettime_nsec_np(CLOCK_MONOTONIC_RAW);

Log("Sleeping async...");
ivars->dispatchSource->WakeAtTime(kIOTimerClockMonotonicRaw, currentTime + fiveSecor
```

After the driver stores the callback, the client app can perform multiple simulated callbacks with the Async Action. This calls `NullDriver::HandleAsyncRequest`, which is largely similar to the delayed call performed in the previous listing.

Important

The driver must register the callback function before the client makes an asynchronous request, or the kernel may panic.

See Also

External drivers

`com.apple.developer.driverkit.userclient-access`

An array of strings that represent macOS driver extensions that may communicate with other DriverKit services.