Mapping IOKit Methods Exposed to User Space on macOS

AUTHOR: Karol Mazurek (@karmaz95) of AFINE

Table of Contents

0 - Introduction

IOKit is the core framework macOS uses for communication between user space and the kernel, exposing numerous driver interfaces. Despite ongoing efforts to harden the platform, IOKit continues to be a frequent source of vulnerabilities. Identifying which methods are accessible from user space is a first step for vulnerability research in this area. It enables the accurate enumeration of all available endpoints, ensuring complete coverage during fuzz testing.

This article outlines a structured methodology for mapping the IOKit external methods exposed to user space. By employing a combination of static analysis and runtime enumeration, we can identify accessible interfaces, pinpoint potential attack vectors, and establish a solid foundation for effective fuzzing. Our goal is to enhance the precision and effectiveness of IOKit vulnerability research.

This guide focuses solely on "external methods," but it is important to note that IOKit drivers also provide other communication channels, such as:

- Properties: For reading and writing driver configuration values
- Notifications: For receiving asynchronous events from the driver
- Shared memory: For efficient large data transfers
- External traps: A legacy method (use external methods instead)
- Shared Data Queue: For bidirectional queued data transfer
- IOStream: For continuous data streaming

While this guide does not cover these additional channels, familiarizing yourself with the material presented here will help improve your understanding of them.

1 - IOKit Interface Fundamentals

The first part of the guide teaches about the main components of the IOKit. It introduces IOKit kernel space components and how to interact with them from User Space.

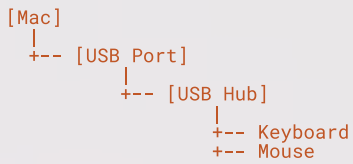
1.0 - Introduction to IOKit

IOKit is a framework in macOS that allows user-space applications to interact with hardware devices, forming part of the XNU kernel. It provides C++ classes and APIs for device drivers, abstracting hardware for easier management. The framework is documented extensively in Apple's IOKit

Fundamentals[0].

1.1 - Registry

Apple's IOKit maintains an IORegistry[1] of all devices in the system, organized as a tree structure. Each node (I/O Service instance) in this tree represents a device, driver, or attachment point, and the relationships between nodes reflect how devices are physically or logically connected. Here is an example of device family tree where a Mac machine has a USB port, a USB hub is connected to that port, and both a keyboard and mouse are plugged into the hub:



Each item ("Mac", "USB Port", etc.) is an IORegistryEntry[2] object (or an object derived from IORegistryEntry), forming the hierarchical structure of the IORegistry.

1.1.1 - Planes

IOKit's IORegistry organizes all services in a tree structure that can be viewed through different "planes". Each plane[3] represents a distinct type of relationship or hierarchy among the same set of objects. The previous example of the USB family tree illustrates the Service plane. It should be the primary plane for mapping the attack surface in IOKit during vulnerability research, as it serves as the root plane. Focusing on other planes may result in missing services.

1.1.2 - Services

The IOService[4] is the base class for all drivers in IOKit, representing hardware devices, virtual devices, or kernel-only system services. These services operate with full kernel privileges and direct hardware access.

User space applications cannot directly instantiate, access, or call methods on I/O service objects.

1.1.3 - Driver

Drivers are classes that inherit from superclasses, all ultimately deriving from the IOService class. For example, MyUSBDriver inherits from IOService, allowing dynamic interaction with it as an IOService object:

```
class MyUSBDriver : public IOService {
public:
    virtual bool init(OSDictionary *dictionary = NULL) override;
    virtual bool start(IOService *provider) override;
    virtual void stop(IOService *provider) override;
    IOReturn sendCommand(uint8_t cmd, uint32_t value);
private:
    IOUSBDevice *fDevice;
};
```

The key methods (init, start, stop) are lifecycle hooks invoked by IOKit as the driver is loaded, initialized, and terminated. Drivers on macOS are Kernel Extensions (KEXTS[5]). The driver is activated using OSKext::start()[6]. In real life, USB drivers are often a subclass of IOUSBDevice or IOUSBInterface, not directly an IOService. The provider parameter in start() is a pointer to the parent object in the IORegistry tree (often a device nub such as IOUSBDevice). The fDevice is a reference to the hardware device or logical service the driver manages.

1.1.4 - Matching

Matching in IOKit refers to the process of finding and loading the appropriate driver for a detected device or service. When a new device, such as a USB device, is detected, IOKit creates a nub (for example, an IOUSBDevice object) in the IORegistry. IOKit then searches for drivers whose matching dictionaries (IOKitPersonalities[7]) specify compatibility with the nub, using a three-phase process:

Class matching: eliminates drivers whose IOProviderClass[8] does not match the nub's class

Passive matching : checks the remaining drivers' personalities for properties in KEXT Info.plist (e.g., vendor, product ID) to further narrow the candidates.

Active matching : for each remaining candidate, it calls the driver's probe()[9] method to verify compatibility and assign a score actively.

```
[IORegistryEntry] (base class)
   |
   +-- [IOService] (abstract service/driver class)
         |
         +-- [IOUSBDevice] (nub created for the USB device)
               |
               +-- [MyUSBDriver] (driver matched and attached to the device)
```

The driver with the highest score is started and attached to the nub, forming the provider-client relationship in the IORegistry tree.

1.1.5 - IOKit Personalities

The most important for us is that we can use the matching APIs to find the services we want to enumerate or fuzz — more on that in "1.3.1". Yet, to do that, we need to know the name under which the service is registered in the IORegistry. These can be found in IOKitPersonalities. Each key is a potential service name, for instance, under IOClass or IOProviderClass:

```
<key>IOKitPersonalities</key>
<dict>
    <key>MyUSBDriver</key>
    <dict>
        <key>CFBundleIdentifier</key>
        <string>com.example.driver.MyUSBDriver</string>
        <key>IOClass</key>
        <string>MyUSBDriver</string>
        <key>IOProviderClass</key>
        <string>IOUSBDevice</string>
        <key>idVendor</key>
        <integer>0x05AC</integer>
        <key>idProduct</key>
        <integer>0x1234</integer>
    </dict>
</dict>
```

A single driver can have multiple personalities, enabling support for different device types or hardware variants without needing separate drivers.

1.1.6 - Service Instances

Not all IOKit personality entries result in instantiated services. A personality defined in a driver's Info.plist will only be matched and loaded if the hardware or software conditions are met. For example, on a Mac mini without an external monitor, any display-related personalities will not be matched, and the corresponding services will not appear in the IORegistry. It's also common for a single driver to have multiple instantiated services when the associated hardware appears more than once, such as with multiple monitors or input devices. Tools like ioscan[10] canbe used to list all instantiated services and often reveal multiple entries with the same service name, such as IOThunderboltPort or IONetworkStack:

```
IOThunderboltPort
IOThunderboltSwitchType5
IOThunderboltPort
IONetworkStack
```

To interact with or fuzz a driver, at least one instance of the corresponding service must be active in the IORegistry, allowing access to the driver's code. It is discussed further in section "2.1.2".

1.2 - User Clients

The IOUserClient[11] is a subclass of IOService that serves as a secure bridge between user-space applications and kernel I/O service objects.

It does not interact with hardware directly but provides a controlled interface for safe communication with kernel services. IOUserClient objects serve as security gatekeepers, running in kernel space and handling requests from unprivileged user-space applications. They validate input, enforce access controls, and sanitize data before passing it to services.

The core logic is implemented in newUserClient[12] functions.

1.2.1 - Multiple User Clients per Service

A single Service can have multiple User Clients registered. They can expose different interfaces to the same underlying Service. Each User Client type has a unique numeric identifier (uint32_t). Applications specify which User Client type they want when connecting with IOServiceOpen()[13]. Type 0 is typically the default/primary interface.

1.2.2 - External Methods

The externalMethod[14] within the User Client handles incoming IOConnectCallMethod[15] requests from the user-space app. It validates that the selector is within bounds and argument sizes. Based on the selector value, it routes them to appropriate handler functions. These are the final endpoints where core logic and most of the vulnerabilities lie.

1.2.3 - Dispatch Mechanism

At WWDC22, the validation portion of the external method was moved to a new "2022" dispatchExternalMethod, which serves as a wrapper around the method array (sIOExternalMethodArray):

```
IOReturn AppleJPEGDriverUserClient::externalMethod(
    IOUserClient* userClient,             // this user client instance
    uint32_t selector,                    // method id from user space
    IOExternalMethodArguments* arguments  // I/O parameters from user space
)
{
    return IOUserClient2022::dispatchExternalMethod(
        userClient,               // the user client object
        selector,                 // which method to call (0-9)
        arguments,                // parameters from user space
        &sIOExternalMethodArray,  // dispatch table with 10 methods
        10,                       // number of methods in table
        userClient,               // target object for method calls
        0                         // additional flags/options
    );
}
```

The dispatchExternalMethod()[16] validates the selector is within bounds, calculates the dispatch table entry at methodArray[selector], checks arguments sizes (I/O scalars and structs), optionally validates entitlements for privileged operations, and finally, call the target handler function if all checks pass.

1.2.4 - getTargetAndMethodForIndex

Although many UserClient::externalMethods were rewritten to include IOUserClient2022: :dispatchExternalMethods, there is still a significant amount of code that follows the old method[17]. There are also getTargetAndMethodForIndex[18]. Drivers using the old way are more prone to misuse or missing validation.

1.2.5 - Access Control

The same selector can mean different things in different User Clients:

```
IOService "MyDevice"
|- IOUserClient Type 0 (standard interface)
|  |- Selector 0: GetStatus
|  `- Selector 1: SetConfig
`- IOUserClient Type 1 (admin interface)
   |- Selector 0: FactoryReset
   `- Selector 1: UpdateFirmware
```

This design provides both functional separation and security boundaries - unprivileged apps receive limited user clients, while privileged ones receive full-featured ones. The entitlements[19] embedded in the application's code signature[20] define access to these interfaces.

1.3 - User Space App

User-space applications can be sandboxed, restricting their access to IOKit even if they possess the necessary entitlements. Sandboxed apps are generally restricted from performing sensitive operations, such as opening hardware service connections or modifying properties. On macOS Sequoia, the Sandbox Operations affecting IOKit access include[21]:

```
- "iokit*""iokit-get-properties"
- "iokit-issue-extension"
- "iokit-open*"
- "iokit-open-user-client"
- "iokit-open-service"
- "iokit-set-properties"
```

The key permission required for a sandboxed app to use IOConnectCallMethod is "iokit-open-user-client." Note that unsandboxed malware does not have such restrictions, and the remainder of the article discusses the context of an unsandboxed app.

1.3.1 - Service Discovery

User space applications can use matching dictionaries to find services based on properties like IOProviderClass, IONameMatch[22], or custom attributes. IOServiceGetMatchingServices()[23] searches the registry and returns matching IOService objects:

```
CFDictionaryRef matching =
IOServiceMatching("IOService");


result = IOServiceGetMatchingServices(masterPort,
matching, &iterator);
```

However, this method iterates only over the root services that directly inherit from the IOService class. To explore deeper into the hierarchy and access all services, use a recursive iterator:

```
io_iterator_t iter;
io_service_t service_ref;

// Create an iterator for the IOService plane, recursively
const kern_return_t kr = IORegistryCreateIterator(
    kIOMainPortDefault,
    kIOServicePlane,
    kIORegistryIterateRecursively,
    &iter
);

// Iterate over every service in the plane
while ((service_ref = IOIteratorNext(iter)) != MACH_PORT_NULL) {
```

Let's say we want to find driver named "NS_01", we can use IORegistryEntryGetName[24]:

```
    char name_buf[128];
    if (IORegistryEntryGetName(service_ref, name_buf) == KERN_SUCCESS) {
        if (strcmp(name_buf, "NS_01") == 0) {
            // Found target service - use service_ref for IOServiceOpen()
            found_service_ref = service_ref;
            break;
        }
    }
    IOObjectRelease(service_ref);
}
```

This provides a service handle ready for communication.

1.3.2 - Spawning User Client

Once a target service is located, we can use IOServiceOpen()[25] to create an IOUserClient instance for communication:

```
io_connect_t connection;
kern_return_t result = IOServiceOpen(
    found_service_ref,     // service from discovery
    mach_task_self(),      // current task
    0,                     // user client type
    &connection            // returned connection handle
);
```

The service validates the request and instantiates the appropriate IOUserClient subclass, returning a connection handle for method calls. On the kernel side, this is handled by SERVICE_NAME::newUserClient functions.

1.3.3 - Calling External Method

Finally, we can use IOConnectCallMethod() to invoke the functionality we want through the established connection. Although there is no direct kernel memory access, this exposure can still introduce vulnerabilities that may lead to kernel code execution[26].

```
uint64_t input_scalar = 0x1234;
uint64_t output_scalar = 0;
uint32_t output_count = 1;

result = IOConnectCallMethod(
    connection,        // connection handle
    5,                 // selector (method index)
    &input_scalar,     // scalar inputs
    1,                 // scalar input count
    NULL,              // struct input buffer
    0,                 // struct input size
    &output_scalar,    // scalar outputs
    &output_count,     // scalar output count
    NULL,              // struct output buffer
    NULL               // struct output size
);
```

The output from the external method, if any, is received through the structure of the output buffer and scalar outputs, while the status code is stored in the result. It's important to note that IOConnectCallMethod is the most commonly used function; however, there are other similar methods, all of which begin with IOConnectCall*[27].

1.4 - User Space Application Flow

```
Service Discovery
    |
    v
+--------------------+     IOServiceMatching()
| Matching Dict      |---> IOServiceGetMatchingServices()
| "DriverName"       |     or IORegistryCreateIterator()
+--------------------+
    |
    v
+--------------------+     IOIteratorNext()
| Service Iterator   |---> IORegistryEntryGetName()
| Loop through all   |     strcmp(name, "target")
+--------------------+
    |
    v
Connection Establishment
    |
    v
+--------------------+     IOServiceOpen()
| io_service_t       |---> Creates IOUserClient
| (Service Handle)   |     Returns io_connect_t
+--------------------+
    |
    v
Method Invocation
    |
    v
+--------------------+     IOConnectCallMethod()
| io_connect_t       |---> Selector + Arguments
| Connection Handle  |     Routed to externalMethod()
+--------------------+

User App  -->  I/O Registry  -->  IOService  -->  IOUserClient
   |               ^                 ^               |
   |               |                 |               |
   +-- Discovery --+                 +-- Bridge -----+
   |                                                 |
   +------------- IOConnectCallMethod ---------------+
```

2 - IOKit Reconeiscance

The second part of the guide shows how to map the attack surface properly to have a complete picture of External Methods exposed from Kernel Space to User Space.

2.0 - What to Map

We need data from IOKit drivers, which enable targeted fuzzing and facilitate faster crash analysis. The result of this entire process should include a structured YAML (or your preferred format) file and a corpus directory with binary files for IOConnectCallMethod.

Fields of interests:

- **Bundle Names:** Kext identifiers containing the driver code. Example: com.apple.iokit.IOAVBFamily
- **Driver Names:** IORegistry service names, so we can match them before using IOServiceOpen. Example: IOAVBNub
- **NewUserClients:** Methods that handle new user client creation. Example: IOAVBNub::newUserClient
- **Types:** Valid `type` values for IOServiceOpen.

- **External Methods:** externalMethod selectors exposed by each user client.
- **Arguments:** Valid scalar and struct sizes for input/output.
- **Endpoints:** Selector IDs per UC, each mapped to argument sizes.
- **Corpus:** Binary samples for inputStruct used by the system or known to be valid.

Each driver maps to its connection types, which map to selector IDs with their argument layout:

```
DriverName:
    TypeValue:
      SELECTOR_ID: [INPUT_SCALAR_CNT, INPUT_STRUCT_SIZE, OUTPUT_SCALAR_CNT,
OUTPUT_STRUCT_SIZE]
```

Example YAML Output Format:

```
AFKEndpointInterface:      # AFKEPInterfaceKextV2::newUserClient
    1768910955:            # AFKEndpointInterfaceUserClient::externalMethod
        0: [2, 10, 16, 10] # extOpenMethod
        1: [1, 10, 1, 10]  # extCloseMethod
        2: [7, 0, 0, 0]    # extEnqueueCommandMethod
```

Corpus Directory Layout:

Each selector gets a directory containing valid binary payloads. These can be passed directly into IOConnectCallMethod as inputStruct payloads.

```
AFKEndpointInterface/1768910955
    |- 0/
    |  `- corpus_0.bin
    |- 1/
    |  `- corpus_0.bin
    `- ...
```

This is a suggested structure for the IOKit MAP. Feel free to organize the data as you like; it's just how I categorize information for my fuzzer.

2.1 - KEXT Analysis

Editor's Note: This paper was truncated for our print release.

Read the full version on phrack.org!