12

MIC AND WEBCAM MONITOR

In "Shut Up and Dance," a poignant episode of the TV show *Black Mirror*, hackers infect a young teenager's computer with malware, spy on him through his webcam, then blackmail him into performing criminal acts. Coincidentally, shortly before the episode aired, I found myself reverse engineering an intriguing piece of Mac malware known as FruitFly that did something very similar.[1]

This persistent backdoor had many capabilities, including the ability to spy on its victims' webcams by leveraging archaic QuickTime APIs. Although these APIs activated a camera's LED indicator light, the malware had a rather insidious trick up its sleeve to attempt to remain undetected; it waited until the victim was inactive before triggering the spying logic. As a result, the victim likely didn't notice that their webcam had been surreptitiously activated.

My investigation of the malware intersected with an FBI operation that led to the arrest of the alleged creator and revealed FruitFly's insidious reach.

According to a Justice Department press release and indictment, the creator had installed FruitFly on thousands of computers over the course of 13 years.[2]

Apple eventually took steps to mitigate this threat, such as creating XProtect detection signatures. Even so, FruitFly remains a stark reminder of the very real dangers Mac users can face, despite Apple's best efforts. FruitFly isn't even the only Mac malware that spies on its victims through the webcam. Others include Mokes, Eleanor, and Crisis.

To address these threats, I released OverSight, a utility that monitors a Mac's built-in mic and webcam, as well as any external connected audio and video devices, and alerts the user about any unauthorized access. In this chapter, I'll explain how OverSight monitors these devices. I'll also demonstrate how this tool ingests system log messages filtered via custom predicates to identify the process responsible for the device access.

You can find OverSight's full source code in the Objective-See GitHub repository at *https://github.com/objective-see/OverSight*.

Tool Design

In a nutshell, OverSight alerts the user whenever their Mac's mic or webcam activates and, most importantly, identifies the responsible process. Thus, whenever malware such as FruitFly attempts to access the camera or mic, this action will trigger an OverSight alert. While OverSight doesn't attempt to classify the process as benign or malicious by design, it provides options for users to either allow or block the process or to exempt trusted processes (Figure 12-1).

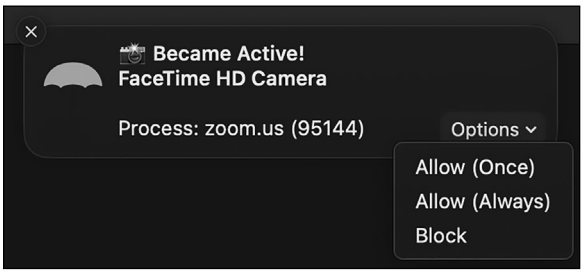

Figure 12-1: OverSight provides the option to always allow a certain tool to access the mic and webcam.

The Allow (Once) option essentially takes no action, as OverSight receives notifications once the device activation has already occurred. However, the Allow (Always) option provides a simple way for users to create rules that keep trusted processes, such as FaceTime or Zoom, from generating alerts in the future. Finally, the Block option will terminate the process by sending it a kill signal (`SIGKILL`).

Compared to tools such as BlockBlock, which contains various components and XPC communications, OverSight is relatively simple. It's a

self-contained, stand-alone app able to perform its mic and webcam monitoring duties with standard user privileges. Let's explore exactly how OverSight achieves this monitoring and, more importantly, identifies the responsible process. We'll see that the former is easy thanks to various `CoreAudio` and `CoreMediaIO` APIs, while the latter is a more challenging task.

Mic and Camera Enumeration

To receive a notification that a process has activated or deactivated each connected mic or webcam, OverSight adds to each device what is known as a property listener for the "is running somewhere" property, `kAudioDevicePropertyDeviceIsRunningSomewhere`. Because the APIs to add such a listener require a device ID, let's first look at how we can enumerate mic and camera devices and then extract each device's ID.

The `AVFoundation`[3] class `AVCaptureDevice`[4] exposes the class method `devicesWithMediaType:`, which takes a media type as an argument (Listing 12-1). To enumerate audio devices such as mics, we use the constant `AVMediaTypeAudio`. To enumerate video devices, we use `AVMediaTypeVideo`. The method returns an array of `AVCaptureDevice` objects that match the specified media type.

```
#import <AVFoundation/AVCaptureDevice.h>

for(AVCaptureDevice* audioDevice in [AVCaptureDevice devicesWithMediaType:AVMediaTypeAudio]) {
    printf("audio device: %s\n", audioDevice.description.UTF8String);

    // Add code here to add a property listener for each audio device.
}
for(AVCaptureDevice* videoDevice in [AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo]) {
    printf("video device: %s\n", videoDevice.description.UTF8String);

    // Add code here to add a property listener for each video device.
}
```

Listing 12-1: Enumerating all audio and video devices

Compiling and running the code in Listing 12-1 outputs the following on my system, which shows my Mac's built-in microphone and webcam and also a pair of connected headphones:

```
Audio device: <AVCaptureHALDevice: 0x11b36a480 [MacBook Pro
Microphone][BuiltInMicrophoneDevice]>

Audio device: <AVCaptureHALDevice: 0x11a7e0440 [Bose QuietComfort 35]
[04-52-C7-77-0D-4E:input]>

Video device: <AVCaptureDALDevice: 0x10dbb2c00 [FaceTime HD Camera]
[3F45E80A-0176-46F7-B185-BB9E2C0E82E3]>
```

You can access the device's name, such as FaceTime HD Camera, in the `localizedName` property of each `AVCaptureDevice` object. You may also want to make use of other object properties such as `modelID`, `manufacturer`, and

deviceType to monitor only a subset of devices. For example, you might choose to monitor only devices built into your Mac.

Audio Monitoring

To set a property listener on each audio device so you can receive activation and deactivation notifications, OverSight implements a helper method named watchAudioDevice: that takes a pointer to an AVCaptureDevice object. For each device of type AVMediaTypeAudio, OverSight invokes this helper.

At the core of this method is a call to the AVFoundation AudioObjectAddPropertyListenerBlock function, defined in the AVFoundation *AudioHardware.h* header file as follows:

```
extern OSStatus AudioObjectAddPropertyListenerBlock(AudioObjectID inObjectID,
const AudioObjectPropertyAddress* inAddress, dispatch_queue_t __nullable inDispatchQueue,
AudioObjectPropertyListenerBlock inListener);
```

The first parameter is an ID for the audio object, for which we can register a property listener. Each AVCaptureDevice object has an object property named connectionID containing this required ID, but it isn't publicly exposed. This means we can't access it directly by writing code such as audioDevice.connectionID. However, as noted elsewhere in this book, you can access private properties either by extending the object's definition or by using the performSelector:withObject: method.

OverSight uses the latter approach. You'll find the logic to obtain the private device ID from an AVCaptureDevice object in a helper method named getAVObjectID: (Listing 12-2).

```
-(UInt32)getAVObjectID:(AVCaptureDevice*)device {
    UInt32 objectID = 0;

  ❶ SEL methodSelector = NSSelectorFromString(@"connectionID");
    if(YES != [device respondsToSelector:methodSelector]) {
        goto bail;
    }

  ❷ #pragma clang diagnostic push
    #pragma clang diagnostic ignored "-Wpointer-to-int-cast"
    #pragma clang diagnostic ignored "-Warc-performSelector-leaks"
  ❸ objectID = (UInt32)[device performSelector:methodSelector withObject:nil];
  ❹ #pragma clang diagnostic pop

bail:
    return objectID;
}
```

Listing 12-2: Obtaining a device's private ID

In Objective-C, you can access object properties, including private ones, by invoking a method on the object that matches the property's name. You can refer to these methods, or indeed any methods, by their names using

selectors. Represented by the SEL type, Objective-C selectors are really just pointers to strings that represent the name of the method. In Listing 12-2, you can see that the code first creates a selector for the connectionID property using the NSSelectorFromString API ❶.

Because connectionID is a private property, nothing is stopping Apple from renaming it or removing it altogether. For that reason, the code invokes the respondsToSelector: method to make sure it's still found on the AVCaptureDevice object; if not, it bails. You should always make use of the respondsToSelector: method before attempting to access private properties or invoking private methods; otherwise, your program risks crashing with a doesNotRecognize Selector exception.[5]

Next, the code makes use of various #pragma directives to save the diagnostic state and tell the compiler to ignore warnings that would otherwise be shown ❷. These warnings get raised when we invoke the performSelector:withObject: method ❸, as the compiler has no way of knowing what object it returns and thus can't know how to manage its memory.[6] Because the connectionID is just an unsigned 32-bit integer, it doesn't need memory management.

Finally, the code accesses the connectionID property via the selector created earlier. It accomplishes this in the aforementioned performSelector: withObject: method, which allows you to invoke an arbitrary selector on an arbitrary object. With the device's identifier in hand, the helper function restores the previous diagnostic state ❹ and returns the device's ID to the caller.

The second argument to the AudioObjectAddPropertyListenerBlock function is a pointer to an AudioObjectPropertyAddress structure, which identifies the property we're interested in receiving a notification about. OverSight initializes the structure, as shown in Listing 12-3.

```
AudioObjectPropertyAddress propertyStruct = {0};
propertyStruct.mSelector = kAudioDevicePropertyDeviceIsRunningSomewhere;
propertyStruct.mScope = kAudioObjectPropertyScopeGlobal;
propertyStruct.mElement = kAudioObjectPropertyElementMain;
```

Listing 12-3: Initializing an AudioObjectPropertyAddress structure

We specify that we're interested in the kAudioDevicePropertyDeviceIs RunningSomewhere property, which relates to device activation and deactivation by any process on the system. The other elements of the structure indicate that the property we specified applies globally to the entire device, not just to a particular input or output. As a result, once we've added the property listener block, OverSight will receive notifications when the specified audio device's run state changes.

The function's third argument is a standard dispatch queue on which to execute the listener block (described next). We can either create a dedicated queue via the dispatch_queue_create API or use dispatch_get_global _queue, for example, with the DISPATCH_QUEUE_PRIORITY_DEFAULT constant, to make use of an existing global queue. The final argument to the function is a block of type AudioObjectPropertyListenerBlock that the Core Audio

framework will automatically invoke whenever the specified property changes on the specified device. Here is the listener block's type definition, also found in *AudioHardware.h*:

```
typedef void (^AudioObjectPropertyListenerBlock)(UInt32 inNumberAddresses,
const AudioObjectPropertyAddress* inAddresses);
```

As multiple properties could change all at once if specified to receive notifications, the listener block gets invoked with an array of `AudioObjectPropertyAddress` objects and the number of elements in this array. OverSight is only interested in a single property, so it ignores these parameters. For completeness, Listing 12-4 shows OverSight's `watchAudioDevice:` method, which contains the core logic for specifying the property of interest, defining a listener block for notifications, and then adding it to the specified audio device.

```
-(BOOL)watchAudioDevice:(AVCaptureDevice*)device {
    AudioObjectPropertyAddress propertyStruct = {0};

    propertyStruct.mSelector = kAudioDevicePropertyDeviceIsRunningSomewhere;
    propertyStruct.mScope = kAudioObjectPropertyScopeGlobal;
    propertyStruct.mElement = kAudioObjectPropertyElementMain;

    AudioObjectID deviceID = [self getAVObjectID:device];

    AudioObjectPropertyListenerBlock listenerBlock =
    ^(UInt32 inNumberAddresses, const AudioObjectPropertyAddress* inAddresses) {
        // Code to handle device's run state changes removed for brevity
    };

    AudioObjectAddPropertyListenerBlock(deviceID, &propertyStruct, self.eventQueue,
    listenerBlock);
    ...
}
```

Listing 12-4: Setting up a listener block for an audio device's run state changes

The OverSight code in the listener block queries the device to determine its current state, as the notification tells us that the run state changed, but not to what state. If it finds the audio device turned on, OverSight consults its log monitor to determine the identity of the process responsible for accessing and activating the device. This step, discussed in more detail in "Responsible Process Identification" on page 288, is unfortunately necessary, because although Apple provides APIs to receive notifications about the state changes of an audio device, they provide no information about the responsible process. Lastly, the listener block alerts the user, providing information about the audio device, its state, and, in activation cases, the responsible process.

To determine whether the device was activated or deactivated, OverSight invokes the `AudioDeviceGetProperty` API within a helper method it names `getMicState:` (Listing 12-5).

```
-(UInt32)getMicState:(AVCaptureDevice*)device {
    UInt32 isRunning = 0;
    UInt32 propertySize = sizeof(isRunning);

    AudioObjectID deviceID = [self getAVObjectID:device]; ❶
    AudioDeviceGetProperty(deviceID, 0, false, kAudioDevicePropertyDeviceIsRunningSomewhere,
    &propertySize, &isRunning); ❷

    return isRunning;
}
```

Listing 12-5: Determining the current state of an audio device

After declaring a few necessary variables, this method invokes the `getAVObjectID:` helper method discussed earlier to extract the private device ID from the `AVCaptureDevice` object that triggered the notification ❶. It then passes this value, along with the `kAudioDevicePropertyDeviceIsRunningSomewhere` constant, a size, and an out pointer for the result, to the `AudioDeviceGetProperty` function ❷. As a result of this call, we'll know whether the notification we received in the callback block occurred due to a device activation or a less interesting deactivation.

Next, I'll show you how to monitor video devices, such as the built-in webcam.

Camera Monitoring

To detect the run-state changes of video devices, which are of type `AVMediaTypeVideo`, we can follow an approach similar to the audio device monitoring code. However, we'll use APIs in the *CoreMediaIO* framework and register a property listener with the `CMIOObjectAddPropertyListenerBlock` API.

OverSight monitors video devices for run-state changes in its `watchVideoDevice:` method (Listing 12-6).

```
-(BOOL)watchVideoDevice:(AVCaptureDevice*)device {
  ❶ CMIOObjectPropertyAddress propertyStruct = {0};
    propertyStruct.mScope = kAudioObjectPropertyScopeGlobal;
    propertyStruct.mElement = kAudioObjectPropertyElementMain;
    propertyStruct.mSelector = ❷ kAudioDevicePropertyDeviceIsRunningSomewhere;

  ❸ CMIOObjectID deviceID = [self getAVObjectID:device];

  ❹ CMIOObjectPropertyListenerBlock listenerBlock = ^(UInt32
    inNumberAddresses, const CMIOObjectPropertyAddress addresses[]) {
        // Code to handle device's run-state changes removed for brevity
    };

  ❺ CMIOObjectAddPropertyListenerBlock(deviceID, &propertyStruct,
    self.eventQueue, listenerBlock);
    ...
}
```

Listing 12-6: Setting up a listener block for a video device's run-state changes

As when monitoring audio devices, the code initializes a property structure to specify the property for which we're interested in receiving notifications ❶. Notice that we use the same constants as for audio devices ❷. Apple's header files don't appear to define a video device–specific constant.

Next, we get the video device's ID using OverSight's `getAVObjectID:` helper method ❸. We also implement a listener block of type `CMIOObject PropertyListenerBlock` ❹, then invoke the `CMIOObjectAddPropertyListenerBlock` function ❺. Once we've made this call, the *CoreMediaIO* framework will automatically invoke the listener block whenever a monitored video device activates or deactivates.

As with audio devices, we must manually query the device to learn whether it was activated or deactivated. You can find this logic in OverSight's `getCameraState:` method, which uses `CoreMediaIO` APIs but is otherwise nearly identical to the `getMicState:` method. As such, I won't cover it here.

Device Connections and Disconnections

So far, we've enumerated the audio and video devices currently connected to the system. For each device, we've added a property listener block that will receive a notification whenever the device activates or deactivates. This is all well and good, but we also need to handle cases in which currently monitored devices disconnect and reconnect, as well as situations in which a user plugs in a new device during the monitoring. For example, imagine that the user regularly connects or disconnects their laptop to an Apple Cinema display. These displays have built-in webcams that OverSight should monitor for unauthorized activations, so we must be able to handle devices that come and go.

Luckily, this is relatively straightforward thanks to the macOS `NS NotificationCenter` dispatch mechanism. Part of the *Foundation* framework, it allows clients to register themselves as observers for events of interest, then receive notifications whenever these events occur. To learn about audio or video device connections and disconnections, we'll subscribe to the events `AVCaptureDeviceWasConnectedNotification` and `AVCaptureDeviceWasDisconnected Notification`, which we can register with the code in Listing 12-7.

```
[NSNotificationCenter.defaultCenter addObserver:self
selector:@selector(handleConnectedDeviceNotification:)
name:AVCaptureDeviceWasConnectedNotification object:nil];

[NSNotificationCenter.defaultCenter addObserver:self
selector:@selector(handleDisconnectedDeviceNotification:)
name:AVCaptureDeviceWasDisconnectedNotification object:nil];
```

Listing 12-7: Registering for device connections and disconnections

OverSight makes two calls to the `addObserver:selector:name:object:` method to register itself for the events of interest. Let's take a closer look at the arguments passed to this method. First is the object, or *observer*, used to handle the notification. OverSight specifies `self` to indicate that the object

registering for the notifications is the same as the object that will handle them. As the second argument, OverSight uses the `@selector` keyword to specify the name of the method to invoke on the observer object and handle the notification. For new device connections, we use an OverSight method named `handleConnectedDeviceNotification:`, and for disconnections, we use `handleDisconnectedDeviceNotification:`. We'll look at these methods shortly.

Next, we specify the event of interest, such as device connection or disconnection. The constants for these events can be found in Apple's *AVCaptureDevice.h* file. The last argument allows you to specify an additional object to deliver along with the notification. OverSight doesn't make use of this and, as such, simply passes `nil`.

Once OverSight has invoked `addObserver:selector:name:object:` twice, whenever a device connects or disconnects, the notification center will invoke our corresponding observer method. The single parameter it passes to this method is a pointer to an `NSNotification` object. In the case of device connection or disconnection, this object contains a pointer to the `AVCaptureDevice`.

Both notification observer methods first extract the device from the notification object and then determine its type (audio or video). Next, the code invokes OverSight's device type–specific methods to either start or stop the monitoring, depending on whether the device was connected or disconnected.

As an example, Listing 12-8 shows the implementation of the `handleConnectedDeviceNotification:` method.

```
-(void)handleConnectedDeviceNotification:(NSNotification *)notification {
  ❶ AVCaptureDevice* device = notification.object;

  ❷ if(YES == [device hasMediaType:AVMediaTypeAudio]) {
        [self watchAudioDevice:device];
  ❸ } else if(YES == [device hasMediaType:AVMediaTypeVideo]) {
        [self watchVideoDevice:device];
    }
}
```

Listing 12-8: When a new device connects, OverSight will begin monitoring it for run-state changes.

The method extracts the device that triggered the notification by accessing the `object` property of the `NSNotification` object passed into it ❶. If this just-connected device is an audio device, the code invokes OverSight's `watchAudioDevice:` method, discussed earlier, to register a property listener block for state changes ❷. For video devices, the code invokes the `watchVideoDevice:` method ❸. The method to handle device disconnections is identical, except it invokes the relevant OverSight *unwatch* methods, discussed in "Stopping" on page 293, which stop the monitoring of audio or video devices.

If we were solely interested in the fact that a video or audio device had activated or deactivated, we'd be done. However, these events have limited

utility for malware detection if they don't include the process responsible for triggering it. So, we have more work cut out for us.

Responsible Process Identification

Many legitimate activities could activate your mic or camera (for example, hopping on a conference call). A security tool must be able to identify the process accessing a device so it can ignore the ones it trusts and generate alerts for any it doesn't recognize.

In previous chapters, I mentioned that Endpoint Security APIs can identify the process responsible for many events of interest. Unfortunately, Endpoint Security doesn't report on mic and camera access yet (although I've begged Apple to add this feature many a time). While we've shown that the `CoreAudio` and `CoreMediaIO` APIs can provide notifications about changes to a device's run state, they don't contain information about the responsible process.

Over the years, OverSight has taken various roundabout approaches to accurately identify the responsible process. Initially, it took advantage of the fact that frameworks within processes accessing the mic or webcam would send various Mach messages to the core macOS camera and audio assistant daemons. When it received a device run-state change notification, OverSight would enumerate any Mach message senders. It also supplemented this information by extracting responsible candidate processes from the I/O registry.[7] Unfortunately, even this combined approach often yielded more than one candidate process. So, OverSight executed the macOS `sample` utility, which provided stack traces of the candidate processes. By examining these stack traces, it could identify whether a process was actively interacting with an audio or video device.

This approach wasn't the most efficient (and the `sample` utility is a touch invasive, as it briefly suspends the target process), but it could consistently identify the responsible process. At the time, OverSight was the only tool on the market able to provide this feature, making it a hit not only with users but also with commercial entities, who reverse engineered the tool to steal this capability for their own purposes—bugs and all! When I confronted the companies with proof of this transgression, all eventually admitted fault, apologized, and made amends.[8]

NOTE *Interestingly, one of the developers who copied OverSight's proprietary logic began working for Apple shortly thereafter. Coincidentally or not, more recent versions of macOS now alert you when a process initially attempts to access the mic or camera. As they say, imitation is the sincerest form of flattery.*

As macOS changed, OverSight's initial method of identifying the responsible process began to show its age. Luckily the introduction of the universal log provides a more efficient solution. In Chapter 6, I showed how to use the universal log's private APIs and frameworks for ingesting streaming log messages, among other tasks. OverSight uses these same APIs and frameworks, coupled with custom filter predicates, to identify the process responsible for triggering any mic or camera state changes.

NOTE *Messages in the log can change at any time. In this section, I focus on the messages present in macOS 14 and 15. While future versions of the operating system could replace these messages, you should be able to identify the new ones and swap them in.*

The universal log contains many messages continually streaming from all corners of the system. To identify relevant messages (for example, those pertaining to processes accessing the camera), let's start a log stream, then fire up an application such as FaceTime that makes use of the webcam:

```
% log stream
...
Default     0x0   367    0    com.apple.cmio.registerassistantservice:
[com.apple.cmio:] RegisterAssistantService.m:2343:-[RegisterAssistantServer
addRegisterExtensionConnection:]_block_invoke [{private}901][{private}0]
added <private> endpoint <private> camera <private>

Default     0x0   901    0    avconferenced: (CoreMediaIO) [com.apple.cmio:]
CMIOHardware.cpp:747:CMIODeviceStartStream backtrace 0   CoreMediaIO
0x000000019b4c4040 CMIODeviceStartStream + 228    [0x19b45a000 + 434240]
```

In the stream, you can see messages related to the camera access. These contain references to a process with the PID of 901 or emanating from that process. In this example, that PID maps to the process `avconferenced`, which accesses the webcam on behalf of FaceTime. Let's try another application (say, Zoom) to see what shows up in the logs:

```
% log stream
...
Default     0x0   367    0    com.apple.cmio.registerassistantservice:
[com.apple.cmio:] RegisterAssistantService.m:2343:-[RegisterAssistantServer
addRegisterExtensionConnection:]_block_invoke [{private}17873][{private}0]
added <private> endpoint <private> camera <private>

Default     0x0   17873  0    zoom.us: (CoreMediaIO) [com.apple.cmio:]
CMIOHardware.cpp:747:CMIODeviceStartStream backtrace 0   CoreMediaIO
0x00007ff8248a6287 CMIODeviceStartStream
+ 205    [0x7ff824840000 + 418439]CMIOHardware.cpp:747:CMIODeviceStartStream
backtrace 0   CoreMediaIO      0x00007ff8248a6287 CMIODeviceStartStream +
205    [0x7ff824840000 + 418439]
```

We receive the exact same messages, except this time they contain a process ID of 17873, which belongs to Zoom. You can perform a similar experiment to identify log messages containing information about processes accessing the mic.

To programmatically interact with the universal log, OverSight implements a custom class named `LogMonitor`. The code in this class interfaces with APIs found within the private *LoggingSupport* framework. Since Chapter 6 covered this strategy, I won't repeat the detail here. If you're interested in the full code, take a look at the *LogMonitor.m* file in the *OverSight* project.

OverSight's `LogMonitor` class exposes a method with the definition shown in Listing 12-9.

```
-(BOOL)start:(NSPredicate*)predicate level:(NSUInteger)level
callback:(void(^)(OSLogEvent*))callback;
```

Listing 12-9: `LogMonitor`'s method to start a log stream filtered by a specified level and predicate

Given a predicate and a log level (such as default or debug), this method activates a streaming log session. It will pass log messages of type `OSLogEvent` that match the specified predicate to the caller using the specified callback block.

OverSight uses a predicate that matches all log messages from either the core media I/O subsystem or the core media subsystem, because these subsystems generate the specific log messages that contain the PID of the responsible process (Listing 12-10).

```
if(@available(macOS 14.0, *)) {
    [self.logMonitor start:[NSPredicate predicateWithFormat:@"subsystem=='com.apple.cmio' OR
    subsystem=='com.apple.coremedia'"] level:Log_Level_Default callback:^(OSLogEvent*
    logEvent) {
        // Code that processes cmio and coremedia log messages removed for brevity
    }];
}
```

Listing 12-10: Filtering messages from the `cmio` and `coremedia` subsystems

We intentionally leave these predicates broad to ensure that macOS performs the predicate matching within the system log daemon's instance of the logging framework, rather than in the instance of the same framework loaded in OverSight. This avoids the significant overhead of copying and transmitting all system log messages between the two processes. The only downside to using a broader predicate is that OverSight must then filter out irrelevant messages. As neither of the two specified subsystems generates a significant number of log messages, however, this additional processing doesn't introduce much overhead.

For each message from the subsystems, OverSight checks whether it contains the PID of the process that triggered the device's run-state change. Listing 12-11 shows the code to do this for camera events.

```
❶ NSRegularExpression* cameraRegex = [NSRegularExpression
  regularExpressionWithPattern:@"\\[\\{private\\}(\\d+)\\]"
  options:0 error:nil];

❷ if( (YES == [logEvent.subsystem isEqual:@"com.apple.cmio"]) &&
      (YES == [logEvent.composedMessage hasSuffix:@"added <private>
      endpoint <private> camera <private>"]) ) {
    ❸ NSTextCheckingResult* match = [cameraRegex firstMatchInString:logEvent.
      composedMessage options:0 range:NSMakeRange(0, logEvent.composedMessage.
      length)];
      if( (nil == match) || (NSNotFound == match.range.location) ) {
```

```
        return;
    }
  ❹ NSInteger pid = [[logEvent.composedMessage substringWithRange:
    [match rangeAtIndex:1]] integerValue];
        self.lastCameraClient = pid;
}
```

Listing 12-11: Parsing `cmio` messages to detect the responsible process

For camera events, we look for a message from the `com.apple.cmio` subsystem ending with `added <private> endpoint <private> camera <private>` ❷. To extract the PID for this process, OverSight uses a regular expression, which it initializes prior to the message processing to avoid reinitialization ❶, then applies it to the candidate messages ❸. If the regular expression doesn't match, the callback exits with a return statement. Otherwise, it extracts the PID as an integer and saves it into an instance variable named `lastCameraClient` ❹. OverSight references this variable when it receives a camera run-state change notification and builds an alert to show the user (Listing 12-12).

```
Client* client = nil;

if(0 != self.lastCameraClient) {
    client = [[Client alloc] init];
    client.pid = [NSNumber numberWithInteger:self.lastCameraClient];
    client.path = valueForStringItem(getProcessPath(client.pid.intValue));
    client.name = valueForStringItem(getProcessName(client.path));
}
Event* event = [[Event alloc] init:client device:device deviceType:
Device_Camera state:NSControlStateValueOn];

[self handleEvent:event];
```

Listing 12-12: Creating an object encapsulating the responsible process

For mic events, the approach is similar, except OverSight looks for messages from the `com.apple.coremedia` subsystem that start with `-MXCoreSession-[MXCoreSession beginInterruption]` and end with `Recording = YES> is going active`.

Using the universal log to identify processes responsible for mic and camera access has proven effective. The strategy's main downside is that Apple occasionally changes or removes relevant log messages. For example, OverSight used different log messages to identify responsible processes in earlier versions of macOS, forcing me to update the tool when Apple removed them. You can see these updates by viewing the *AVMonitor.m* commit history in OverSight's GitHub repository.

Triggering Scripts

When I introduced OverSight in 2015, macOS provided no restrictions on mic or webcam access, meaning any malware that infected the system could trivially access either. Recent versions of macOS have addressed this

shortcoming by prompting the user the first time any application attempts to access these devices. Unfortunately, this approach relies on the operating system's Transparency, Consent, and Control (TCC) mechanism, which hackers and malware often bypass, as noted in Chapter 6.

Besides providing an additional layer of defense, OverSight offers features that users have leveraged creatively. For example, it provides a mechanism to take additional actions whenever a process accesses the mic or camera. If you open OverSight's preferences and click the Action tab, you'll see that you can specify a path to an external script or binary. If a user provides such an executable, OverSight will execute it upon each activation event.

To further enhance this capability, another option allows users to enable arguments to provide to the script, including the device, state, and responsible process. This makes OverSight relatively easy to integrate into other security tools (although users have frequently used the feature for more practical reasons, such as turning on an external light outside their home office whenever they activate their mic or camera).

OverSight's code to execute external scripts or binaries is fairly straightforward, though the handling of arguments requires a few nuances. OverSight makes use of the `NSUserDefaults` class to persistently store settings and preferences, including any user-specified script or binary. Listing 12-13 shows the code that saves the path of an item when the user interacts with the Browse button.

```
#define PREF_EXECUTE_PATH @"executePath"
#define PREF_EXECUTE_ACTION @"executeAction"

❶ self.executePath.stringValue = panel.URL.path;
...
❷ [NSUserDefaults.standardUserDefaults setBool:NSControlStateValueOn
  forKey:PREF_EXECUTE_ACTION];

❸ [NSUserDefaults.standardUserDefaults setObject:self.executePath.stringValue
  forKey:PREF_EXECUTE_PATH];

❹ [NSUserDefaults.standardUserDefaults synchronize];
```

Listing 12-13: The `NSUserDefaults` class used to store user preferences

We save the path of the item the user selected via the user interface ❶, then set a flag indicating that the user specified an action ❷ and save the item's path ❸. Note that `panel` is an `NSOpenPanel` object containing the item the user selected. We set the flag using the `setBool:` method of the `NSUserDefaults`'s `standardUserDefaults` object and set the item path using the `setObject:` method. Finally, we synchronize to trigger a save ❹.

When the user specifies an external item to run, OverSight invokes a helper function named `executeUserAction:` to run the item when a run-state change occurs to a mic or camera (Listing 12-14).

```
#define SHELL @"/bin/bash"
#define PREF_EXECUTE_PATH @"executePath"
```

```
#define PREF_EXECUTE_ACTION_ARGS @"executeActionArgs"

-(BOOL)executeUserAction:(Event*)event {
    NSMutableString* args = [NSMutableString string];

    NSString* action = [NSUserDefaults.standardUserDefaults objectForKey:PREF_EXECUTE_PATH]; ❶
    if(YES == [NSUserDefaults.standardUserDefaults boolForKey:PREF_EXECUTE_ACTION_ARGS]) { ❷
        [args appendString:@"-device "]; ❸
        (Device_Camera == event.deviceType) ? [args appendString:@"camera"] :
        [args appendString:@"microphone"];

        [args appendString:@" -process "];
        [args appendString:event.client.pid.stringValue];
        ...
    }

  ❹ execTask(SHELL, @[@"-c", [NSString stringWithFormat:@"\"%@\" %@", action, args]], NO, NO);
    ...
```

Listing 12-14: Executing a user-specified item with arguments

The `executeUserAction:` method first extracts the path of the user-specified item to execute from the saved preference ❶. Then it checks whether the user has opted to pass arguments to the item ❷. If so, it dynamically builds a string containing the arguments, including the device that triggered the event and the responsible process ❸. Finally, it executes the item and any arguments via the shell using the `execTask` helper function ❹ discussed in previous chapters.

You might be wondering why OverSight executes the user-specified item via */bin/bash* instead of just executing the item directly. Well, as the shell supports the execution of both scripts and stand-alone executables, this means users can specify either in OverSight.

Stopping

It's nice to provide users with an easy way to pause or fully disable a security tool they have installed. I'll end this chapter by looking at OverSight's code to stop the device and log monitor. I won't cover the UI components and logic that expose this ability, but you can find them implemented as a macOS status bar menu in OverSight's *Application/StatusBarItem.m* file.

When a user disables or stops OverSight, it first stops its log monitor by calling a `stop` method that the custom log monitor exposes. This method ends the stream that ingests log messages by invoking the `OSLogEventLiveStream` object's `invalidate` method. Once the log monitor has stopped, OverSight stops monitoring all audio and video devices in two loops (Listing 12-15).

```
-(void)stop {
    ...
    for(AVCaptureDevice* audioDevice in [AVCaptureDevice devicesWithMediaType:AVMediaType
    Audio]) {
        [self unwatchAudioDevice:audioDevice];
    }

    for(AVCaptureDevice* videoDevice in [AVCaptureDevice devicesWithMediaType:AVMediaType
    Video]) {
        [self unwatchVideoDevice:videoDevice];
    }
    ...
}
```

Listing 12-15: Ending the monitoring of all devices

One loop iterates over all audio devices, calling OverSight's `unwatchAudioDevice:` method, and a second loop iterates over video devices to invoke `unwatchVideoDevice:` on them. The code in these methods, which remove listener blocks, is nearly identical to the `watch*` monitoring methods covered earlier in this chapter, as you can see in this snippet from the `unwatchAudioDevice` method (Listing 12-16).

```
-(void)unwatchAudioDevice:(AVCaptureDevice*)device {
    ...
    AudioObjectID deviceID = [self getAVObjectID:device];

    AudioObjectPropertyAddress propertyStruct = {0};
    propertyStruct.mScope = kAudioObjectPropertyScopeGlobal;
    propertyStruct.mElement = kAudioObjectPropertyElementMain;
    propertyStruct.mSelector = kAudioDevicePropertyDeviceIsRunningSomewhere;

  ❶ AudioObjectRemovePropertyListenerBlock(deviceID,
    &propertyStruct, self.eventQueue, self.audioListeners[device.uniqueID]);
    ...
}
```

Listing 12-16: Removing a property listener block from an audio device

The code in this listing first gets the specified device's ID and then initializes an `AudioObjectPropertyAddress` that describes the previously added property listener ❶. It passes these, along with the listener block stored in the dictionary named `audioListeners`, to the `AudioObjectRemovePropertyListenerBlock` function. This fully removes the property listener block, ending OverSight's monitoring of the device.

Conclusion

Some of the most insidious threats targeting Mac users spy on their victim using the mic or camera. Instead of trying to detect specific malware specimens, OverSight counters all of them by taking the simple, albeit powerful, heuristic-based approach of detecting unauthorized mic and camera access.

In this chapter, I first showed you how OverSight leverages various `CoreAudio` and `CoreMediaIO` APIs to register for notifications about mic and camera activations and deactivations. Then we explored the tool's use of a custom log monitor to identify the process responsible for the event. Finally, I showed you how users can easily extend OverSight to execute external scripts or binaries as it detects events and the logic behind stopping OverSight.

In the next chapter, we'll continue to explore the building of robust security tools by looking at how to create a DNS monitor capable of detecting and blocking unauthorized network access.

Notes

1. Selena Larson, "Mac Malware Caught Silently Spying on Computer Users," CNN Money, July 24, 2017, *https://money.cnn.com/2017/07/24/technology/mac-fruitfly-malware-spying/index.html*.
2. US Department of Justice, Office of Public Affairs, "Ohio Computer Programmer Indicted for Infecting Thousands of Computers with Malicious Software and Gaining Access to Victims' Communications and Personal Information," press release no. 18-21, January 10, 2018, *https://www.justice.gov/opa/pr/ohio-computer-programmer-indicted-infecting-thousands-computers-malicious-software-and*.
3. "AVFoundation," Apple Developer Documentation, *https://developer.apple.com/documentation/avfoundation?language=objc*.
4. "AVCapture Device," Apple Developer Documentation, *https://developer.apple.com/documentation/avfoundation/avcapturedevice?language=objc*.
5. "doesNotRecognizeSelector:," Apple Developer Documentation, *https://developer.apple.com/documentation/objectivec/nsobject/1418637-doesnotrecognizeselector?language=objc*.
6. "performSelector May Cause a Leak Because Its Selector Is Unknown," Stack Overflow, November 18, 2018, *https://stackoverflow.com/a/20058585*.
7. "The I/O Registry," Apple Documentation Archive, last updated April 9, 2014, *https://developer.apple.com/library/archive/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/TheRegistry/TheRegistry.html*.
8. You can read more about this series of events in Corin Faife, "This Mac Hacker's Code Is So Good, Corporations Keep Stealing It," The Verge, August 11, 2022, *https://www.theverge.com/2022/8/11/23301130/patrick-wardle-mac-code-corporations-stealing-black-hat*.