6

LOG MONITORING

If you've spent time poking around macOS, you may have encountered the system's unified logging mechanism, a resource that can help you understand macOS internals and, as you'll soon see, uncover malware. In this chapter, I'll start by highlighting the various kinds of information that can be extracted from these logs to detect malicious activity. We'll then reverse engineer the macOS `log` utility and one of its core private frameworks so we can programmatically ingest real-time information directly and efficiently from the logging subsystem.

Exploring Log Information

I'll begin by covering a few examples of useful activity that can show up in the system log, starting with webcam access. Especially insidious malware specimens, including FruitFly, Mokes, and Crisis, surreptitiously spy on their victims through the infected host's webcam. Accessing the webcam generates system log messages, however. For example, depending on the version of macOS, the Core Media I/O subsystem may produce the following:

```
CMIOExtensionProvider.m:2671:-[CMIOExtensionProvider setDevicePropertyValuesForClientID:
deviceID:propertyValues:reply:] <CMIOExtensionProvider>,
3F4ADF48-8358-4A2E-896B-96848FDB6DD5, propertyValues {
    CMIOExtensionPropertyDeviceControlPID = 90429;
}
```

The bolded value contains the ID of the process accessing the webcam. Although the process could be legitimate, such as a Zoom or FaceTime session launched by the user for a virtual meeting, it's prudent to confirm that this is the case, as the responsible process could also be malware attempting to spy on the user. Because Apple doesn't provide an API that identifies the process accessing the webcam, log messages are one of the only ways to reliably get this information most of the time.

Other activities that often show up in system logs are remote logins, which could indicate a compromise, such as attackers gaining initial access to a host or even returning to a previously infected one. For example, the IPStorm malware spreads to victims by brute-forcing SSH logins.[1] Another interesting case is XCSSET, which locally initiates a seemingly remote connection back to the host to bypass the macOS security mechanism known as Transparency, Consent, and Control (TCC).[2]

When a remote login occurs via SSH, the system generates log messages such as the following:

```
sshd: Accepted keyboard-interactive/pam for Patrick from 192.168.1.176 port 59363 ssh2
sshd: (libpam.2.dylib) in pam_sm_setcred(): Establishing credentials
sshd: (libpam.2.dylib) in pam_sm_setcred(): Got user: Patrick
...
sshd: (libpam.2.dylib) in pam_sm_open_session(): UID: 501
sshd: (libpam.2.dylib) in pam_sm_open_session(): server_URL: (null)
sshd: (libpam.2.dylib) in pam_sm_open_session(): path: (null)
sshd: (libpam.2.dylib) in pam_sm_open_session(): homedir: /Users/Patrick
sshd: (libpam.2.dylib) in pam_sm_open_session(): username: Patrick
```

These log messages provide the source IP address of the connection, as well as the identity of the user who logged in. This information can help defenders determine whether the SSH session is legitimate (perhaps a remote worker connecting to their office machine) or unauthorized.

Log messages can also provide insight into the TCC mechanism, which governs access to sensitive information and hardware features. In an Objective by the Sea conference talk, "The Clock Is TCCing," researchers

Calum Hall and Luke Roberts noted that messages found in the unified log enabled them to determine several pieces of information for a given TCC event (for example, malware attempting to capture the screen or access a user's documents), including the resource for which the process requested access, the responsible and target processes, and whether the system denied or approved the request and why.[3]

At this point, it may be tempting to treat log messages as a panacea for malware detection. Don't. Apple doesn't officially support log messages and has often changed their contents or removed them altogether, even between minor releases of macOS. For example, on older versions of the operating system, you could detect microphone access and identify the process responsible for it by looking for the following log message:

```
send: 0/7 synchronous to com.apple.tccd.system: request: msgID=408.11,
function=TCCAccessRequest, service=kTCCServiceMicrophone, target_token={pid:23207, auid:501,
euid:501},
```

Unfortunately, Apple updated the relevant macOS framework so it no longer produces the message. If your security tool relied solely on this indicator to detect unauthorized microphone access, it would no longer function. Thus, it's best to treat log messages as initial signs of suspicious behavior, then investigate further.

The Unified Logging Subsystem

We often think of log messages as a way to figure out what happened in the past. But macOS also lets you subscribe to the stream of messages as they're delivered to the logging subsystem in essentially real time. Better yet, the logging subsystem supports the filtering of these messages via custom predicates, providing efficient and unparalleled insight into the activity happening on the system.

In versions of macOS beginning with 10.12, this logging mechanism is called the *unified logging system*.[4] A replacement of the traditional syslog interface, it records messages from core system daemons, operating system components, and any third-party software that generates logging messages via the `OSLog` APIs.

It's worth noting that if you examine log messages in the unified system log, you may encounter redactions; the logging subsystem replaces any information deemed sensitive with the string `<private>`. To disable this functionality, you could install a configuration profile.[5] While useful for understanding undocumented features of the operating system, however, you shouldn't disable log redactions on end-user or production systems, which would make sensitive data available to anybody with access to the log.

Manually Querying the log Utility

To manually interface with the logging subsystem, use the macOS `log` utility found in */usr/bin*:

```
% /usr/bin/log
usage:
    log <command>

global options:
    -?, --help
    -q, --quiet
    -v, --verbose

commands:
    collect         gather system logs into a log archive
    config          view/change logging system settings
    erase           delete system logging data
    show            view/search system logs
    stream          watch live system logs
    stats           show system logging statistics

further help:
    log help <command>
    log help predicates
```

You can search previously logged data with the `show` flag or use the `stream` flag to view logging data as it's generated in real time. Unless you specify otherwise, the output will include messages with a default log level only. To override this setting for past data, use the `--info` or `--debug` flag, along with `show`, to view further information and debug messages, respectively. For streaming data, specify both `stream` and `--level`, then either `info` or `debug`. These flags are hierarchical; specifying the debug level will return informational and default messages too.

Use the `--predicate` flag with a predicate to filter the output. A rather extensive list of valid predicate fields allows you to find messages based on the process, subsystem, type, and much more. For example, to stream log messages from the kernel, execute the following:

```
% log stream --predicate 'process == "kernel"'
```

There is often more than one way to craft a predicate. For instance, we could also receive kernel messages by using `'processIdentifier == 0'`, as the kernel always has a process ID of 0.

To stream messages from the security subsystem, enter the following:

```
% log stream --predicate 'subsystem == "com.apple.securityd"'
```

The examples shown here all use the equality operator (`==`). However, predicates can use many other operators, including comparative operators

(such as ==, !=, and <), logical operators (such as AND and OR), and even membership operators (such as BEGINSWITH and CONTAINS). Membership operators are powerful, as they allow you to craft filter predicates resembling regular expressions.

The log man pages and the command log help predicates provide a succinct overview of predicates.[6]

Reverse Engineering log APIs

To read log data programmatically, we could use the OSLog APIs.[7] These APIs return only historical data, however, and in the context of malware detection, we're much more interested in real-time events. No public API allows us to achieve this, but by reverse engineering the log utility (specifically, the code that backs the stream command), we can uncover exactly how to ingest logging messages as they enter the unified logging subsystem. Moreover, by providing a filter predicate, we can receive only messages of interest to us.

Although I won't cover the full details of reversing the log utility, I'll provide an overview of the process in this section. Of course, you could apply a similar process against other Apple utilities and frameworks to extract private APIs useful for malware detection (as we showed in Chapter 3 while implementing package code signing checks).

First, we need to find the binary that implements the logging subsystem's APIs so we can invoke them from our own code. Normally, we'll find such APIs in a framework that is dynamically linked into the utility's binary. By executing otool with the -L command line option, we can view the frameworks against which the log utility is dynamically linked:

```
% otool -L /usr/bin/log
/System/Library/PrivateFrameworks/ktrace.framework/Versions/A/ktrace
/System/Library/PrivateFrameworks/LoggingSupport.framework/Versions/A/LoggingSupport
/System/Library/PrivateFrameworks/CoreSymbolication.framework/Versions/A/CoreSymbolication
...
```

Based on its name, the *LoggingSupport* framework seems likely to contain relevant logging APIs. In past versions of macOS, you could find the framework in the */System/Library/PrivateFrameworks/* directory, while in newer versions, you'll find it in the shared *dyld* cache.

After loading the framework into Hopper (which can directly load frameworks from the *dyld* cache), we find that the framework implements an undocumented class named OSLogEventLiveStream whose base class is OSLogEventStreamBase. These classes implement methods such as activate, setEventHandler:, and setFilterPredicate:. We also encounter an undocumented OSLogEventProxy class that appears to represent log events. Here are some of its properties:

```
NSString* process;
int processIdentifier;
NSString* processImagePath;
```

```
NSString* sender;
NSString* senderImagePath;
NSString* category;
NSString* subsystem;
NSDate* date;
NSString* composedMessage;
```

By examining the `log` utility, we can see how it uses these classes and their methods to capture streaming log data. For example, here is a decompiled snippet from the `log` binary:

```
r21 = [OSLogEventLiveStream initWithLiveSource:...];
[r21 setEventHandler:&var_110];
...
[r21 setFilterPredicate:r22];

printf("Filtering the log data using \"%s\"\n", @selector(UTF8String));
...
[r21 activate];
```

In the decompilation, we first see a call to `initWithLiveSource:` initializing an `OSLogEventLiveStream` object. Calls to methods such as `setEventHandler:` and `setFilterPredicate:` then configure this object, stored in the `r21` register. After the predicate is set, a helpful debug message indicates that a provided predicate can filter log data. Finally, the object activates, which triggers the ingestion of streaming log messages matching the specified predicate.

Streaming Log Data

Using the information we gleaned by reverse engineering the `log` binary and *LoggingSupport* framework, we can craft code to directly stream data from the universal logging subsystem in our detection tools. Here, we'll cover important parts of the code, though you're encouraged to consult the full code, found in this chapter's *logStream* project.

Listing 6-1 shows a method that accepts a log filter predicate, a log level (such as `default`, `info`, or `debug`), and a callback function to invoke for each logging event that matches the specified predicate.

```
#define LOGGING_SUPPORT @"/System/Library/PrivateFrameworks/LoggingSupport.framework"

-(void)start:(NSPredicate*)predicate
level:(NSUInteger)level eventHandler:(void(^)(OSLogEventProxy*))eventHandler {
    [[NSBundle bundleWithPath:LOGGING_SUPPORT] load]; ❶
    Class LiveStream = NSClassFromString(@"OSLogEventLiveStream"); ❷

    self.liveStream = [[LiveStream alloc] init]; ❸

    @try {
        [self.liveStream setFilterPredicate:predicate]; ❹
    } @catch (NSException* exception) {
```

```
        // Code to handle invalid predicate removed for brevity
    }
    [self.liveStream setInvalidationHandler:^void (int reason, id streamPosition) {
        ;
    }];

    [self.liveStream setDroppedEventHandler:^void (id droppedMessage) {
        ;
    }];

    [self.liveStream setEventHandler:eventHandler]; ❺
    [self.liveStream setFlags:level]; ❻

    [self.liveStream activate]; ❼
}
```

Listing 6-1: Starting a logging stream with a specified predicate

Note that I've omitted part of this code, such as the class definition and properties of the custom log class.

After loading the logging support framework ❶, the code retrieves the private `OSLogEventLiveStream` class by name ❷. Now we can instantiate an instance of the class ❸. We then configure this instance by setting the filter predicate ❹, making sure to wrap it in a `try...catch` block, as the `setFilterPredicate:` method can throw an exception if provided with an invalid predicate. Next, we set the event handler, which the framework will invoke anytime the universal logging subsystem ingests a log message matching the specified predicate ❺. We pass these values into the `start:level:eventHandler:` method, where the predicate tells the log stream how to filter the messages it delivers to the event handler. We set the logging level via the `setFlags:` method ❻. Finally, we start the stream with a call to the `activate` method ❼.

Listing 6-2 shows how to create an instance of the custom log monitor class and then use it to begin ingesting log messages.

```
NSPredicate* predicate = [NSPredicate predicateWithFormat:<some string predicate>]; ❶

LogMonitor* logMonitor = [[LogMonitor alloc] init]; ❷

[logMonitor start:predicate level:Log_Level_Debug eventHandler:^(OSLogEventProxy* event) {
    printf("New Log Message: %s\n\n", event.description.UTF8String);
}];

[NSRunLoop.mainRunLoop run];
```

Listing 6-2: Interfacing with the custom log stream class

First, the code creates a predicate object from a string ❶. Note that in production code, you should also wrap this action in a `try...catch` block, as the `predicateWithFormat:` method throws a catchable exception if the provided predicate is invalid. Next, we create a `LogMonitor` object and invoke its `start:level:eventHandler:` method ❷. Note that for the level, we pass in `Log_Level_Debug`. Since the level is hierarchal, this will ensure we capture all

message types, including those whose type is `info` and `default`. Now the code will invoke our event handler anytime a log message matching the specified predicate streams to the universal logging subsystem. Currently, this handler simply prints out the `OSLogEventProxy` object.

To compile this code, we'll need the undocumented class and method definitions we extracted from the *LoggingSupport* framework. These definitions live in the *logStream* project's *LogStream.h* file; Listing 6-3 provides a snippet of them.

```
@interface OSLogEventLiveStream : NSObject
    -(void)activate;
    -(void)setFilterPredicate:(NSPredicate*)predicate;
    -(void)setEventHandler:(void(^)(id))callback;
    ...
    @property(nonatomic) unsigned long long flags;
@end

@interface OSLogEventProxy : NSObject
    @property(readonly, nonatomic) NSString* process;
    @property(readonly, nonatomic) int processIdentifier;
    @property(readonly, nonatomic) NSString* processImagePath;
    ...
@end
```

Listing 6-3: The interface for the private `OSLogEventLiveStream` and `OSLogEventProxy` classes

Once we compile this code, we can execute it with a user-specified predicate. For example, let's monitor the log messages of the security subsystem, *com.apple.securityd*:

```
% ./logStream 'subsystem == "com.apple.securityd"'
New Log Message:
<OSLogEventProxy: 0x155804080, 0x0, 400, 1300, open(%s,0x%x,0x%x) = %d>
New Log Message:
<OSLogEventProxy: 0x155804080, 0x0, 400, 1300, %p is a thin file (%s)>
New Log Message:
<OSLogEventProxy: 0x155804080, 0x0, 400, 1300, %zd signing bytes in %d blob(s) from %s(%s)>
New Log Message:
<OSLogEventProxy: 0x155804080, 0x0, 400, 1009, network access disabled by policy>
```

Although we're indeed capturing streaming log messages that match the specified predicate, the messages don't appear all that useful at first glance. This is because our event handler simply prints out the `OSLogEventProxy` object via a call to its `description` method, which doesn't include all components of the message.

Extracting Log Object Properties

To detect activity that could indicate the presence of malware, you'll want to extract the `OSLogEventProxy` log method object's properties. While disassembling, we encountered several useful properties, such as the process ID, path, and message, but other interesting ones exist as well. Because

Objective-C is introspective, you can dynamically query any object, including undocumented ones, to reveal its properties and values. This requires a foray into the bowels of the Objective-C runtime; nevertheless, you'll find it useful to understand any undocumented classes you encounter, especially when leveraging Apple's private frameworks.

Listing 6-4 is a simple function that accepts any Objective-C object, then prints out its properties and their values. It's based on code by Pat Zearfoss.[8]

```
#import <objc/message.h> ❶
#import <objc/runtime.h>

void inspectObject(id object) {
    unsigned int propertyCount = 0 ;
    objc_property_t* properties = class_copyPropertyList([object class], &propertyCount); ❷

    for(unsigned int i = 0; i < propertyCount; i++) {
        NSString* name = [NSString stringWithUTF8String:property_getName(properties[i])]; ❸

        printf("\n%s: ", [name UTF8String]);

        SEL sel = sel_registerName(name.UTF8String); ❹
        const char* attr = property_getAttributes(properties[i]); ❺

        switch(attr[1]) {
            case '@':
                printf("%s\n",
                [[((id (*)(id, SEL))objc_msgSend)(object, sel) description] UTF8String]);
                break;
            case 'i':
                printf("%i\n", ((int (*)(id, SEL))objc_msgSend)(object, sel));
                break;
            case 'f':
                printf("%f\n", ((float (*)(id, SEL))objc_msgSend)(object, sel));
                break;
            default:
                break;
        }
    }

    free(properties);
    return;
}
```

Listing 6-4: Introspecting the properties of an Objective-C object

First, the code imports the required Objective-C runtime header files ❶. Then it invokes the `class_copyPropertyList` API to get an array and the count of the object's properties ❷. We iterate over this array to examine each property, invoking the `property_getName` method to get the name of the property ❸. Then the `sel_registerName` function retrieves a selector for the property ❹. We'll use the property selector later to retrieve the object's value.

Next, to determine the type of the property, we invoke the `property_getAttributes` method ❺. This returns an array of attributes, with the property type as the second item (at index `1`). The code handles common types such as Objective-C objects (`@`), integers (`i`), and floats (`f`). For each type, we invoke the `objc_msgSend` function on the object with the property's selector to retrieve the property's value.

If you look closely, you'll see that the call to `objc_msgSend` is typecast appropriately for each property type. For a list of type encodings, see Apple's "Type Encodings" developer documentation.[9] To inspect Swift objects, use Swift's Mirror API.[10]

In the log monitor code, we can now invoke the `inspectObject` function with each `OSLogEventProxy` object received from the logging subsystem (Listing 6-5).

```
NSPredicate* predicate = [NSPredicate predicateWithFormat:<some string predicate>];

[logMonitor start:predicate level:Log_Level_Debug eventHandler:
^(OSLogEventProxy* event) {
    inspectObject(event);
}];
```

Listing 6-5: Inspecting each log message, encapsulated in an `OSLogEventProxy` object

If we compile and execute the program, we should now receive a more comprehensive view of each log message. For example, by monitoring messages related to XProtect, the built-in antimalware scanner found on certain versions of macOS, we can observe its scan of an untrusted application:

```
% ./logStream 'subsystem == "com.apple.xprotect"'

New Log Message:

composedMessage: Starting malware scan for: /Volumes/Install/Install.app

logType: 1
timeZone: GMT-0700 (GMT-7) offset -25200
...
processIdentifier: 1374
process: XprotectService
processImagePath: /System/Library/PrivateFrameworks/XprotectFramework
.framework/Versions/A/XprotectService.xpc/Contents/MacOS/XprotectService
...
senderImagePath: /System/Library/PrivateFrameworks/XprotectFramework
.framework/Versions/A/XprotectService.xpc/Contents/MacOS/XprotectService
sender: XprotectService
...
subsystem: com.apple.xprotect
category: xprotect
...
```

The abridged output contains the properties of the `OSLogEventProxy` object most relevant to security tools. Table 6-1 summarizes these alphabetically.

As with many OSLogEventProxy object properties, you can use them in custom predicates.

Table 6-1: Security-Relevant OSLogEventProxy Properties

Property name	Description
category	The category used to log an event
composedMessage	The contents of the log message
logType	For logEvent and traceEvent, the message's type (default, info, debug, error, or fault)
processIdentifier	The process ID of the process that caused the event
processImagePath	The full path of the process that caused the event
senderImagePath	The full path of the library, framework, kernel extension, or Mach-O image that caused the event
subsystem	The subsystem used to log an event
type	The type of event (such as activityCreateEvent, activityTransitionEvent, or logEvent)

Determining Resource Consumption

It's important to consider the potential resource impact of streaming log messages. If you take an overly consumptive approach, you can incur a significant CPU cost and impact to the responsiveness of the system.

First, pay attention to the log level. Specifying the debug level will result in a significant increase in the number of log messages processed against any predicate. Although the predicate evaluation logic is very efficient, more messages mean more CPU cycles. Thus, a security tool that leverages the logging subsystem's streaming capabilities should probably stick to consuming the default or info messages.

Equally important to efficiency is the predicate you use. Interestingly, my experiments have shown that the logging daemon wholly evaluates some predicates, while the logging subsystem frameworks loaded in client programs, such as the log monitor, handle others. The former is better; otherwise, the program will receive a copy of every single log message for predicate evaluation, which can chew up significant CPU cycles. If the logging daemon performs the predicate evaluation, you'll receive messages that match the predicate only, which won't discernibly impact the system.

How can you craft a predicate that the logging daemon will evaluate? Trial and error have shown that if you specify a process or subsystem in a predicate, the daemon will evaluate it, meaning you'll receive only log messages that match. Let's look at a specific example from OverSight, a tool discussed in Chapter 12 that monitors the microphone and webcam.[11]

OverSight requires access to log messages from the core media I/O subsystem to identify the process accessing the webcam. At the start of the chapter, I noted that certain versions of macOS store this process ID in log messages from the core media I/O subsystem that contain the string

`CMIOExtensionPropertyDeviceControlPID`. Understandably, you might be tempted to craft a predicate that matches this string:

```
'composedMessage CONTAINS "CMIOExtensionPropertyDeviceControlPID"'
```

This predicate would lead to processing inefficiencies, however, as the logging daemon will send all messages that the logging frameworks loaded in our log monitor to perform the predicate filtering. Instead, OverSight leverages a broader predicate that makes use of the `subsystem` property:

```
subsystem=='com.apple.cmio'
```

This approach causes the logging daemon to perform the predicate matching, then deliver only messages from the core media I/O subsystem. OverSight itself manually performs the check for the `CMIOExtensionProperty DeviceControlPID` string:

```
if(YES == [logEvent.composedMessage
containsString:@"CMIOExtensionPropertyDeviceControlPID ="]) {
    // Extract the PID of the processes accessing the webcam.
}
```

The tool leverages a similar process to return log messages associated with mic access. As a result, it can effectively detect any process (including malware) attempting to use either the mic or webcam.

Conclusion

In this chapter, you saw how to use code to interface with the operating system's universal logging subsystem. By reverse engineering the private *LoggingSupport* framework, we programmatically streamed messages matching custom predicates and accessed the wealth of data found in the logging subsystem. Security tools could use this information to detect new infections or even uncover the malicious actions of persistently installed malware.

In the next chapter, you'll write network monitoring logic using Apple's powerful and well-documented network extensions.

Notes

1. Nicole Fishbein and Avigayil Mechtinger, "A Storm Is Brewing: IPStorm Now Has Linux Malware," Intezer, November 14, 2023, *https://www.intezer.com/blog/research/a-storm-is-brewing-ipstorm-now-has-linux-malware/.*
2. "The XCSSET Malware," TrendMicro, August 13, 2020, *https://documents.trendmicro.com/assets/pdf/XCSSET_Technical_Brief.pdf.* To read more about the abuse of remote logins in macOS, see Jaron Bradley, "What Does APT Activity Look Like on macOS?," *The Mitten Mac*, November 14, 2021, *https://themittenmac.com/what-does-apt-activity-look-like-on-macos/.*

3. Calum Hall and Luke Roberts, "The Clock Is TCCing," paper presented at Objective by the Sea v6, Spain, October 12, 2023, *https://objectivebythesea.org/v6/talks/OBTS_v6_lRoberts_cHall.pdf*.
4. "Logging," Apple Developer Documentation, *https://developer.apple.com/documentation/os/logging*.
5. Howard Oakley, "How to Reveal 'Private' Messages in the Log," Eclectic Light, May 25, 2020, *https://eclecticlight.co/2020/05/25/how-to-reveal-private-messages-in-the-log/*.
6. See Howard Oakley, "log: A Primer on Predicates," Eclectic Light, October 17, 2016, *https://eclecticlight.co/2016/10/17/log-a-primer-on-predicates/*, and "Predicate Programming Guide," Apple Developer Documentation, *https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Predicates/AdditionalChapters/Introduction.html*.
7. "OSLog," Apple Developer Documentation, *https://developer.apple.com/documentation/oslog*.
8. Pat Zearfoss, "Objective-C Quickie: Printing All Declared Properties of an Object," April 14, 2011, *https://zearfoss.wordpress.com/2011/04/14/objective-c-quickie-printing-all-declared-properties-of-an-object/*.
9. The list is available at *https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtTypeEncodings.html#//apple_ref/doc/uid/TP40008048-CH100-SW1*.
10. Read more about Swift's Mirror API in Antoine van der Lee, "Reflection in Swift: How Mirror Works," *SwiftLee*, December 21, 2021, *https://www.avanderlee.com/swift/reflection-how-mirror-works/*.
11. See *https://objective-see.org/products/oversight.html*.