13

DNS MONITOR

In this chapter, I'll focus on the practicalities of building a deployable host-based network monitor capable of proxying and blocking DNS traffic from unrecognized processes or destined for untrusted domains.

Chapter 7 covered the basic design of a DNS proxy capable of monitoring traffic via Apple's *NetworkExtension* framework. There, however, I skipped over many of the steps required to build a deployable tool, including obtaining necessary entitlements and correctly bundling the extension within a host application. This chapter will discuss these tasks, as well as ways of extending a basic monitor, such as by parsing DNS queries and responses to block those found on a block list.

You can find these capabilities and more in the open source DNSMonitor, which is part of Objective-See's tool suite (*https://github.com/objective-see/DNSMonitor*). I recommend that you download the project or reference the source code in the repository while reading the chapter, as the following discussions often omit parts of the code for brevity.

Network Extension Deployment Prerequisites

Modern networking monitors, including DNSMonitor, make use of the network extension framework. Because they're packaged as system extensions and run as stand-alone processes with elevated privileges, Apple requires developers to entitle and bundle them in a very specific way. In Chapter 11, we walked through the process of obtaining the Endpoint Security entitlement and then creating a provisioning profile for the tool in the Apple Developer portal. If you're building a network extension, you'll follow a similar process, with a few key differences.

First, you'll need to generate two provisioning profiles, one for the network extension and another for the application that contains and loads the extension. Follow the process described in Chapter 11 to create an ID for each item on the Apple Developer site. When asked to select capabilities for the extension, check **Network Extensions**, which maps to the *com.apple.developer.networking.networkextension* entitlement. Any developer can use this entitlement (unlike the Endpoint Security entitlement, which requires explicit approval from Apple). For the application, select that same capability, as well as **System Extension**, which will allow the application to install, load, and manage the extension. Once you've created both IDs, create the two provisioning profiles.

Now you must install each provisioning profile in Xcode. If you look at the *DNSMonitor* project, you'll see that it contains two targets: the extension and its host application. When you click either of these targets, the Signing and Capabilities tab should provide an option to specify the relevant provisioning profile. Apple's developer documentation recommends enabling manual signing by leaving the Automatically Manage Signing option unchecked.[1]

The Signing and Capabilities tab will also show that the DNSMonitor project has enabled additional capabilities for both the extension and application that match those we specified when building the provisioning profile. The extension specifies the Network Extensions capability, while the app specifies both Network Extensions and System Extensions. If you're building your own network extension, you'll have to add these capabilities manually by clicking the + next to Capabilities.

Behind the scenes, adding these capabilities applies the relevant entitlements to each target's *entitlements.plist*. Unfortunately, we must manually edit these *entitlements.plist* files. Adding the Network Extensions capability and checking DNS Proxy will add the entitlement with a value of `dns-proxy`, but we'll need a value of `dns-proxy-systemextension` to deploy an extension signed with a developer ID.[2] Listing 13-1 shows this in the extension's *entitlements.plist* file.

```
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
    <key>com.apple.developer.networking.networkextension</key>
    <array>
        <string>dns-proxy-systemextension</string>
    </array>
    ...
```

Listing 13-1: We must entitle network extensions and specify an extension type.

The file includes the network extension entitlement as a key, along with an array holding any extension types.

Packaging the Extension

Any tool that uses a network extension must implement it as a system extension, then structure itself in a specific way so that macOS can validate and activate it. Specifically, Apple requires that any system extension be packaged within a bundle, such as an application, in the bundle's *Contents/Library/SystemExtensions/* directory. A provisioning profile must also authorize the use of restricted entitlements, and we can't embed provisioning profiles directly into a stand-alone binary.

For these reasons, DNSMonitor contains two components: a host application and a network extension.[3] To properly package the extension in Xcode, we specify the application component dependency on the extension under **Build Phases**. We set the destination to **System Extensions** so that macOS will copy the extension into the application's *Contents/Library/SystemExtensions/* directory while building the application (Figure 13-1).

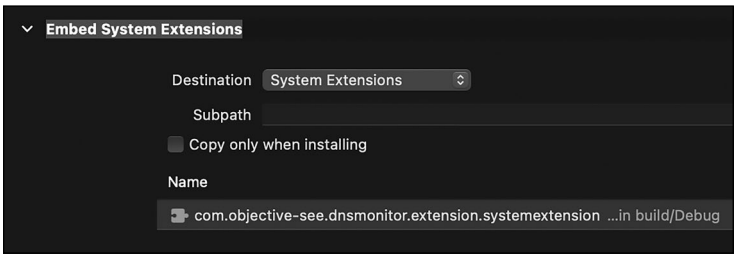

Figure 13-1: The application contains a build step to embed the system extension.

Let's now turn our attention to the extension's *Info.plist* file (Listing 13-2).

```
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
    ...
❶ <key>CFBundlePackageType</key>
```

```
    <string>$(PRODUCT_BUNDLE_PACKAGE_TYPE)</string>
    ...
  ❷ <key>NetworkExtension</key>
    <dict>
      ❸ <key>NEMachServiceName</key>
        <string>$(TeamIdentifierPrefix)com.objective-see.dnsmonitor</string>
      ❹ <key>NEProviderClasses</key>
            <dict>
                <key>com.apple.networkextension.dns-proxy</key>
                <string>DNSProxyProvider</string>
            </dict>
        </dict>
        ...
```

Listing 13-2: The extension's Info.plist *file contains various key-value pairs specific to network extensions.*

We set `CFBundlePackageType` to a variable ❶ that the compiler will replace with the project's type, `systemextension`. The `NetworkExtension` key holds a dictionary containing key and value pairs relevant to network extensions ❷. The `NEMachServiceName` key specifies the name of the Mach service the extension can use for XPC communications ❸. Also, note the `NEProviderClasses` key, which contains the network extension's type and the name of the class within DNSMonitor that implements the required network extension logic ❹. In Chapter 7, I mentioned that this class should implement `NEDNSProxyProvider` delegate methods. We must also link the extension component against the *NetworkExtension* framework.

The application's *entitlements.plist* file, shown in Listing 13-3, is fairly similar to that of the extension.

```
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
    <key>com.apple.developer.networking.networkextension</key>
    <array>
        <string>dns-proxy-systemextension</string>
    </array>
    <key>com.apple.developer.system-extension.install</key>
    <true/>
    <key>com.apple.security.application-groups</key>
    <array>
        <string>$(TeamIdentifierPrefix)com.objective-see.dnsmonitor</string>
    </array>
</dict>
</plist>
```

Listing 13-3: The app's entitlements.plist *file also contains key-value pairs specific to network extensions.*

One difference between the two is the addition of the *com.apple.developer.system-extension.install* entitlement, set to true. We indirectly added this entitlement to the app's provisioning profile when we granted it the System

Extension capability. The app needs this entitlement to install and activate the network extension.

Tool Design

Now that I've explained the components of DNSMonitor, let's focus on how it operates, starting with launching the application.

The App

You can find the initialization logic for the app in the *DNSMonitor/App/main.m* file. After performing some basic argument parsing (for example, checking whether the user invoked the app with the `-h` flag to show the default usage), the app retrieves the responsible parent's bundle ID. If this parent is the Finder or the Dock (the likely parents in scenarios where the user double-clicked the app icon), the app displays an informative alert explaining that DNSMonitor should run from the terminal.

Also, unless we run DNSMonitor from the *Applications* directory, when the `OSSystemExtensionRequest request:didFailWithError:` delegate method is invoked by the application to activate the extension, it will fail:[4]

```
ERROR: method '-[Extension request:didFailWithError:]' invoked with
<OSSystemExtensionActivationRequest: 0x600003a8f150>, Error Domain=
OSSystemExtensionErrorDomain Code=3 "App containing System Extension
to be activated must be in /Applications folder" UserInfo={NSLocalized
Description=App containing System Extension to be activated must be in
/Applications folder}
```

So, when run from the terminal, DNSMonitor checks that it's executing from the correct directory before loading the network extension component. If not, it prints an error message and exits (Listing 13-4).

```
if(YES != [NSBundle.mainBundle.bundlePath hasPrefix:@"/Applications/"]) {
    ...
    NSLog(@"\n\nERROR: As %@ uses a System Extension, Apple requires it must
    be located in /Applications\n\n", [APP_NAME stringByDeletingPathExtension]);
    goto bail;
}
```

*Listing 13-4: Checking whether the monitor is running from the /*Applications *directory*

To pass captured DNS traffic from the extension to the application so we can display it to the user, we use the system log. In Listing 13-5, the application initializes a custom log monitor with a predicate to match messages written to the log by the (soon-to-be-loaded) network extension. It then prints any received messages to the terminal.

```
NSPredicate* predicate =
[NSPredicate predicateWithFormat:@"subsystem='com.objective-see.dnsmonitor'"];

LogMonitor* logMonitor = [[LogMonitor alloc] init];
[logMonitor start:predicate level:Log_Level_Default eventHandler:^(OSLogEventProxy* event) {
    ...
    NSLog(@"%@", event.composedMessage);
}];
```

Listing 13-5: The app's log monitor ingests DNS traffic captured in the extension.

In other cases, you might want to use a more robust mechanism, such as XPC, to pass data back and forth between the extension and the app, but for a simple command line tool, the universal logging subsystem suffices.

Before loading the network extension, the app sets up a signal handler for the interrupt signal (`SIGINT`). As a result, when the user presses CTRL-C, the app can unload the extension and gracefully exit (Listing 13-6).

```
❶ signal(SIGINT, SIG_IGN);

  dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_SIGNAL,
❷ SIGINT, 0, dispatch_get_main_queue());
❸ dispatch_source_set_event_handler(source, ^{
      ...
      stopExtension();
      exit(0);
  });
  dispatch_resume(source);
```

Listing 13-6: Setting up a custom interrupt signal handler

First, the code ignores the default `SIGINT` action ❶. Then it creates a dispatch source for the interrupt signal ❷ and sets a custom handler with the `dispatch_source_set_event_handler` API ❸. The custom handler invokes a helper function, `stopExtension`, to unload and uninstall the network extension before exiting. Though not shown here, the monitor can be executed with a command line option to skip unloading the extension when it exits. This alleviates the need to restart, and thus reapprove, the extension each time the monitor is restarted.

Finally, the app installs and activates the network extension. Because I covered this process in full detail in Chapter 7, I won't repeat it here, other than to note that it involves making an `OSSystemExtensionRequest` request and configuring an `NEDNSProxyManager` object. You can find the full installation and activation code in DNSMonitor's *App/Extension.m* file.

With the network extension running, the app tells the current run loop to continue until it receives an interrupt signal from the user, as it needs to hang around to print out captured DNS traffic.

The Extension

Behind the scenes, when an application invokes the APIs to install and activate a network extension, macOS copies the extension from the app's

Contents/Library/SystemExtensions/ directory into a privileged directory, */Library/SystemExtensions/<UUID>/*, validates it, then executes it with root privileges. Run the **ps** command to show the activated network extension's process information, such as its privilege level, process ID, and path:

```
% ps aux
...
root 38943 ... /Library/SystemExtensions/8DC3FC3A-825E-49C3-879B-6B0C08388238/
com.objective-see.dnsmonitor.extension.systemextension/Contents/MacOS/com
.objective-see.dnsmonitor.extension
```

Once loaded, DNSMonitor's extension opens a handle to the universal logging subsystem via the `os_log_create` API, as it passes captured DNS traffic to the app using log messages. The logging API takes two parameters that allow you to specify a subsystem and a category (Listing 13-7).

```
#define BUNDLE_ID "com.objective-see.dnsmonitor"

os_log_t logHandle = os_log_create(BUNDLE_ID, "extension");
```

Listing 13-7: Opening a log handle in the extension

By specifying a subsystem or a category, you can easily create predicates that return only certain messages, as we did in the application (Listing 13-5). Next, the extension invokes the `NEProvider` class's `startSystemExtensionMode` method, which you'll recall will instantiate the class specified under the `NEProviderClasses` key in the extension's *Info.plist* file. The extension uses its `DNSProxyProvider` class, which inherits from the `NEDNSProxyProvider` class (Listing 13-8).

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

Listing 13-8: The interface for the `DNSProxyProvider` class

In Chapter 7, I described how a DNS monitor could implement the various `NEDNSProxyProvider` methods, such as the all-important `handleNewFlow:`, which will be automatically invoked for all new DNS flows. As such, I won't cover this again here, though you can find the full code in the *Extension/DNSProxyProvider.m* file.

Previous chapters didn't cover how the extension sends the message to the app via the log, builds a DNS cache, and blocks specific requests or responses. Let's explore these topics in more detail.

Interprocess Communication

I mentioned that when DNSMonitor's network extension receives a new DNS request or response, it uses the universal logging subsystem to send the message to the app's log monitor, which prints it to the terminal. You can find the extension logic to handle the writing of DNS traffic to the log in a helper method named `printPacket` (Listing 13-9).

```
-(void)printPacket:(dns_reply_t*)packet flow:(NEAppProxyFlow*)flow {
    ...
    char* bytes = NULL;
    size_t length = 0;

  ❶ NSMutableDictionary* processInfo = [self getProcessInfo:flow];

    os_log(logHandle, "PROCESS:\n%{public}@\n", processInfo);

  ❷ FILE* fp = open_memstream(&bytes, &length);
  ❸ dns_print_reply(packet, fp, 0xFFFF);
  ❹ fflush(fp);

    os_log(logHandle, "PACKET:\n%{public}s\n", bytes);

    fclose(fp);
    free(bytes);
}
```

Listing 13-9: Printing a DNS packet to the universal log

A helper function named `getProcessInfo:` creates a dictionary that describes the process responsible for generating the DNS traffic. The code then writes the dictionary to the log using the `os_log` API ❶.

Writing the bytes of the DNS packet is a bit more complicated, because the macOS `dns_print_reply` API, which formats raw DNS packets, expects to print to a file stream pointer (`FILE *`), such as `stdout`. On the other hand, universal logging APIs take an `os_log_t` instead of a `FILE *`. We circumvent this minor obstacle by having `dns_print_reply` indirectly write to a memory buffer, which we can log via `os_log`.

To make `dns_print_reply` write to a buffer, we pass it a file handle that, unbeknownst to the function, is backed by a buffer, created thanks to the often-overlooked `open_memstream` API ❷. The `dns_print_reply` function formats the raw DNS packet and then happily writes it via the file handle ❸. After invoking `fflush` to ensure all buffered data is written out to the underlying memory ❹, we write the parsed DNS packet to the universal log with a second call to `os_log`. As I previously noted, the log monitor in the app component can now ingest the message and print it to the user's terminal.

Building and Dumping DNS Caches

It always surprises me that macOS doesn't provide a way to dump cached DNS resolutions, which contain the requested domains and resolved IP addresses. As you'll see in this section, however, DNS cache dumping is easy enough to implement in a DNS monitor.

When the DNSMonitor network extension starts, it creates a global array to store dictionaries of the mappings between DNS requests (questions) and their responses (answers). It implements this logic in a helper method named `cache:`, which takes a parsed DNS response packet that contains both the questions and any answers.

The majority of code within the `cache:` method is dedicated to extracting the questions and answers from the DNS response packet, which can contain multiples of both. We covered this process in Chapter 7, so we won't repeat it here, but you can find the method's full code in *Extension/DNSProxyProvider.m*.

Once we've extracted all questions and answers from the DNS response packet, we add them to the global cache array, named `dnsCache` (Listing 13-10).

```
-(void)cache:(dns_reply_t*)packet {
    NSMutableArray* answers = [NSMutableArray array];
    NSMutableArray* questions = [NSMutableArray array];

    // Code to extract questions and answers from DNS response packet removed

  ❶ @synchronized(dnsCache) {
      ❷ if(dnsCache.count >= MAX_ENTRIES) {
            [dnsCache removeObjectsInRange:NSMakeRange(0, MAX_ENTRIES/2)];
        }

      ❸ for(NSString* question in questions) {
            if(0 != answers.count) {
              ❹ [dnsCache addObject:@{question:answers}];
            }
        }
        ...
    }
}
```

Listing 13-10: Saving DNS questions and answers to a cache

As DNS responses can arrive and be processed asynchronously, we synchronize access to the global cache by wrapping it in a `@synchronized` block ❶. Before adding another entry, the code checks that the cache hasn't grown too large. If it has, it rather bluntly prunes the first half to evict the oldest ones ❷. Finally, it adds an entry for each question and its answers ❸ using the `NSMutableArray`'s `addObject:` method. Note that the snippet of code `@{question:answers}` uses the Objective-C shorthand `@{}` to create a dictionary whose key is the question and whose value is a list of answers ❹.

At this point, the extension is caching DNS questions and answers. The entries generated by resolving NoStarch.com and Objective-See.org would look like the following:

```
[
    {nostarch.com:["104.20.120.46", "104.20.121.46"]},
    {objective-see.org:["185.199.110.153", "185.199.109.153",
    "185.199.111.153", "185.199.108.153"]}
]
```

To facilitate the dumping of this cache, the extension installs a signal handler for the signal `SIGUSR1`, otherwise known as *user signal 1* (Listing 13-11).

```
signal(SIGUSR1, dumpDNSCache);
```

Listing 13-11: Installing a signal handler for user signal 1

Now, any adequately privileged process on the system can send a `SIGUSR1` to the extension. Here's how to do this manually in the terminal:

```
% sudo kill -SIGUSR1 `pgrep com.objective-see.dnsmonitor.extension`
```

The `kill` shell command benignly sends a `SIGUSR1` to the extension, whose process ID we find via `pgrep`. Because the extension is running with root privileges, we must elevate our privileges with `sudo` to deliver a signal.

As the code in Listing 13-11 showed, the extension sets the handler for `SIGUSR1` to a function named `dumpDNSCache`. Let's take a look at this function. Shown in Listing 13-12, it straightforwardly writes each cache entry to the universal log.

```
void dumpDNSCache(int signal) {
    for(NSDictionary* entry in dnsCache) {
      ❶ NSString* question = entry.allKeys.firstObject;
      ❷ os_log(logHandle, "%{public}@:%{public}@", question, entry[question]);
    }
    ...
}
```

Listing 13-12: When the code receives a `SIGUSR1` signal, it dumps the cache to the log.

In a `for` loop, the code iterates over all entries in its global DNS cache. Recall that this cache is an array of dictionaries. Each entry's dictionary contains a single key representing the DNS question, and the code extracts it with the `firstObject` property of the `allKeys` array ❶. Then, using `os_log`, it writes the question and the corresponding answers ❷. Note the use of the `public` keyword, which tells the logging subsystem not to redact the cache data being logged.

When you send a `SIGUSR1` to the extension while the DNSMonitor application component is running, it will automatically ingest the log message containing the dumped cache and print it out:

```
Dumping DNS Cache:
DNSMonitor[2027:25144] www.apple.com:(
    "23.2.84.211"
)
DNSMonitor[2027:25144] nostarch.com:(
    "104.20.120.46",
    "104.20.121.46"
)
DNSMonitor[2027:25144] objective-see.org:(
    "185.199.111.153",
    "185.199.110.153",
    "185.199.109.153",
    "185.199.108.153"
)
```

Because the extension writes the items in its cache to the universal log, you can also view these messages directly via the `log` command:

```
% log stream --predicate="subsystem='com.objective-see.dnsmonitor'"
```

I recommend specifying the filter predicate, however, because otherwise, you'll be inundated with irrelevant log messages from the rest of the system.

Blocking DNS Traffic

So far, we've focused on passive actions, such as printing DNS requests and responses and dumping an extension-built cache. But what if we wanted to extend the monitor to block certain traffic? Chapter 7 covered Apple's official way of blocking traffic using a network extension that implements a filter data provider to allow, drop, or pause network flows. Objective-See's open source firewall LuLu takes this approach.[5]

It turns out we can also block DNS traffic using an `NEDNSProxyProvider` object. Because we're already proxying all DNS traffic, nothing stops us from closing any flow we so choose. A benefit of sticking with the `NEDNSProxyProvider` class is that the system routes only DNS traffic through the extension. Because we're not interested in other types of traffic, this keeps our code efficient. On the other hand, a filter data provider would make us responsible for examining and responding to all network flows.

One simple approach to specifying what DNS traffic to block is to use a block list. This block list could contain the domains and IP addresses of known malware command-and-control servers, unscrupulous internet service providers, or even servers that track users or display ads. Whenever an application attempts to resolve a domain, macOS will proxy the request through the extension, which can examine the request and block it if the domain is on the list. On the flip side, once a remote DNS server has processed a request and resolved the domain, macOS will proxy the response back through the extension before sending it to the application that made the original request. This gives the extension a chance to examine the response and block it if it contains a banned IP address.

You can find the logic to block a domain or IP address in the extension, in a method named `shouldBlock:`. This method accepts a parsed DNS packet of type `dns_reply_t` (used for both requests and responses) and returns a Boolean to indicate whether to block it. The method's logic is rather involved, as it must handle both IPv4 and IPv6, so I won't show its entire code here. Listing 13-13 includes the part of the method that checks whether requests contain any domains on the block list.

```
-(BOOL)shouldBlock:(dns_reply_t*)packet {
    BOOL block = NO;
    dns_header_t* header = packet->header;

    if(DNS_FLAGS_QR_QUERY == (header->flags & DNS_FLAGS_QR_MASK)) { ❶
        for(uint16_t i = 0; i < header->qdcount; i++) { ❷
```

```
            NSString* question = [NSString stringWithUTF8String:packet->question[i]->name]; ❸
            if(YES == [self.blockList containsObject:question]) { ❹
                block = YES;
                goto bail;
            }
        }
    }
    ...

bail:
    return block;
}
```

Listing 13-13: Checking for domains to block

The code first initializes a `dns_header_t` pointer to the header of the parsed DNS packet. Defined in Apple's *dns_util.h* file, it contains flags (to indicate the type of DNS packet) and various counts, such as the number of questions and answers:

```
typedef struct {
    uint16_t xid;
    uint16_t flags;
    uint16_t qdcount;
    uint16_t ancount;
    uint16_t nscount;
    uint16_t arcount;
} dns_header_t;
```

The code in Listing 13-13 checks the header's `flags` member to see whether the `DNS_FLAGS_QR_QUERY` bit is set ❶. This flag indicates that the DNS packet is a query containing one or more domains to resolve. (You won't find constants such as `DNS_FLAGS_QR_QUERY` in any header file, as Apple defines them in *dns_util.c*, so you might want to copy them directly into your own code.) Assuming the DNS packet contains a query, the code then iterates over each domain in the request ❷. The number of domains is stored in the `qdcount` member of the header structure, while each domain to be resolved can be found in the packet's `question` array. The code extracts each domain and converts it to a more manageable Objective-C string object ❸ before checking whether it matches any of the items in the global block list ❹. If so, the code sets a flag, breaks, and returns.

Though not shown here, the code to check a response packet is similar. Response packets list the number of answers in the `ancount` member of the header structure and provide the answers themselves in the `answer` array. Apple defines the `dns_resource_record_t` structure to store these answers in the *dns_util.h* header file. This structure contains, among other things, a `dnstype` member, which specifies the answer's type, such as `A` or `CNAME`. So, to extract an IPv4 address from a DNS `A` record into an Objective-C object, you might write code similar to Listing 13-14.

```
if(ns_t_a == packet->answer[i]->dnstype) {
    NSString* address =
    [NSString stringWithUTF8String:inet_ntoa(packet->answer[i]->data.A->addr)];

    // Add code here to process the extracted answer (IP address).
}
```

Listing 13-14: Extracting an answer from a DNS `A` record

If a question or an answer matches an entry in DNSMonitor's global block list, the `shouldBlock:` method returns `YES`, the Objective-C equivalent of true.

The location of the `shouldBlock:` method's invocation dictates how the flow closes. For example, it's easy to block a question, as DNSMonitor is really a proxy that is responsible for making the actual connection to the remote DNS server and thus we can close the local flow using the `closeWriteWithError:` method (Listing 13-15).

```
BOOL block = [self shouldBlock:parsedPacket];
if(YES == block) {
    [flow closeWriteWithError:nil];
    return;
}
```

Listing 13-15: Closing a local flow

To block an answer, we should make sure to also clean up the remote connection with the DNS server that provided the answer (Listing 13-16).

```
nw_connection_receive(connection, 1, UINT32_MAX, ^(dispatch_data_t content,
nw_content_context_t context, bool is_complete, nw_error_t receive_error) {
    ...
    BOOL block = [self shouldBlock:parsedPacket];
    if(YES == block) {
        [flow closeWriteWithError:nil];
        nw_connection_cancel(connection);
        return;
    }
});
```

Listing 13-16: Closing a remote flow

DNSMonitor uses the `nw_connection_receive` API to proxy responses. Thus, to block any responses, it first closes the flow and then calls `nw_connection_cancel` to cancel the connection.

For completeness, I should mention that you could also handle DNS blocking by returning a response with the response code set to what is known as a *name error* or, more simply, `NXDOMAIN`. Such a response would tell the requestor that the domain wasn't found, meaning the resolution failed. DNSMonitor takes this approach when executed with the `-nx` command line option.

To generate such a response, you could take the DNS request or response packet and modify the flags in its header in the manner shown in Listing 13-17.

```
dns_header_t* header = (dns_header_t *)packet.bytes;

header->flags |= htons(0x8000);
header->flags &= ~htons(0xF);
header->flags |= htons(0x3);
```

Listing 13-17: Crafting an `NXDOMAIN` response

The code expects a DNS packet in a mutable data object. It first typecasts the packet's bytes to a `dns_header_t` pointer. Next, it sets the `QR` bit of the `flags` field in the header to indicate that the packet is a response. Following this, it clears the `RCODE` (response code) bits before setting just the `NXDOMAIN` response code. You can read more about the DNS header and these fields in the RFP 1035 that defines the technical specifications of DNS.[6]

Classifying Endpoints

Instead of using a hardcoded block list, a tool could determine whether to block DNS requests or responses heuristically, for example, by examining historical DNS records, WHOIS data, and any SSL/TLS certificates.[7] Let's look at each of these techniques more closely, using the 3CX supply chain attack as an example. The *3cx.cloud* domain used in the attack is a legitimate part of 3CX's infrastructure, but the attacker-controlled *msstorageboxes.com* domain, used by the malicious code introduced into the application, raises some red flags:

Historical DNS records At the time of the 3CX supply chain attack in March 2023, only one DNS record existed for the *msstorageboxes.com* domain, which had been registered just a few months prior. Trusted domains usually have a longer history and many DNS records. On the other hand, hackers often register domains for their command-and-control servers just before their attacks and tear them down shortly thereafter. Of course, hackers sometimes leverage previously legitimate domains that they either bought through standard domain procurement processes or obtained when domain registration lapsed. Again, you'll see this activity reflected in the domain's historical DNS records.

Redacted WHOIS data The attackers redacted WHOIS data for the *msstorageboxes.com* domain for privacy reasons. It's unusual for a large, well-established company to hide its identity. For example, the legitimate 3cx.cloud domain clearly shows that it's registered to 3CX Software DMCC.

Domain name registrar The attackers registered the *msstorageboxes.com* domain via NameCheap. Well-established companies often choose more enterprise-focused domain registrars, such as CloudFlare.

Conclusion

A DNS monitor capable of tracking all requests and responses is a powerful tool for malware detection. In this chapter, I built on Chapter 7 to describe how you might implement such a monitor atop Apple's *NetworkExtension* framework. I showed you how to add capabilities to the tool, such as a cache and blocking capabilities, to extend its functionality.

In the book's final chapter, we'll pit tools such as this DNS monitor against real-life Mac malware. Read on to see how each side fares!

Notes

1. "Network Extensions Entitlement," Apple Developer Documentation, *https://developer.apple.com/documentation/bundleresources/entitlements/com_apple_developer_networking_networkextension*.
2. psichel, "com.apple.developer.networking.networkextension Entitlements Don't Match PP," Apple Developer Forums, November 15, 2020, *https://developer.apple.com/forums/thread/667045*.
3. "Signing a Daemon with a Restricted Entitlement," Apple Developer Documentation, *https://developer.apple.com/documentation/xcode/signing-a-daemon-with-a-restricted-entitlement*.
4. "Installing System Extensions and Drivers," Apple Developer Documentation, *https://developer.apple.com/documentation/systemextensions/installing-system-extensions-and-drivers?language=objc*.
5. See *https://github.com/objective-see/LuLu*.
6. See "Domain Names—Implementation and Specification," RFC 1035, Internet Engineering Task Force, *https://datatracker.ietf.org/doc/html/rfc1035*.
7. Esteban Borges, "How to Perform Threat Hunting Using Passive DNS," *Security Trails*, *https://securitytrails.com/blog/threat-hunting-using-passive-dns*.