9

MUTING AND AUTHORIZATION EVENTS

In the previous chapter, I introduced Apple's Endpoint Security and its notification events. In this chapter, I move into more advanced topics, such as muting, mute inversion, and authorization events.

Muting instructs Endpoint Security to withhold the delivery of certain events, such as those generated from chatty system processes. Conversely, *mute inversion* gives us the ability to create focused tools that, for example, subscribe solely to events from a specific process or only those related to the access of a few directories. Lastly, Endpoint Security's authorization capabilities offer a mechanism to prevent undesirable actions altogether.

You'll find the majority of the code snippets presented in this chapter in the *ESPlayground* project introduced in Chapter 8. For each topic covered here, I'll point to the part of this project where the relevant code resides, as well as how to execute it via command line arguments.

Muting

All event monitoring implementations risk facing an overwhelming deluge of events. For example, file I/O events occur constantly as part of normal system activity, and file monitors may generate so much data that finding events tied to malicious processes becomes quite difficult. One solution is to mute irrelevant processes or paths. For example, you'll likely want to ignore file I/O events involving the temporary directory or originating from certain chatty, legitimate operating system processes (such as the Spotlight indexing service), as these events occur almost constantly and are rarely useful for malware detection.

Luckily for us, Endpoint Security provides a flexible and robust muting mechanism. Its `es_mute_path` function will suppress events either from a specified process or that match a specified path. The function takes three parameters—a client; a path to a process, directory, or file; and a type:

```
es_mute_path(es_client_t* _Nonnull client, const char* _Nonnull path,
es_mute_path_type_t type);
```

The mute path type can be one of the four values found in the enumeration of type `es_mute_path_type_t` in *ESTypes.h*:

```
typedef enum {
    ES_MUTE_PATH_TYPE_PREFIX,
    ES_MUTE_PATH_TYPE_LITERAL,
    ES_MUTE_PATH_TYPE_TARGET_PREFIX,
    ES_MUTE_PATH_TYPE_TARGET_LITERAL
} es_mute_path_type_t;
```

The types ending in `PREFIX` tell Endpoint Security that the path provided to `es_mute_path` is a prefix to a longer path. For example, you could use the `ES_MUTE_PATH_TYPE_TARGET_PREFIX` option to mute all file I/O events originating from a certain directory. On the other hand, if the mute path type ends in `LITERAL`, the path has to match exactly for events to be muted.

Use the initial two values of the enumeration, `ES_MUTE_PATH_TYPE_PREFIX` and `ES_MUTE_PATH_TYPE_LITERAL`, when you want to mute the path of the process responsible for triggering the Endpoint Security event. For example, Listing 9-1 shows a snippet from the `mute` function (in the *ESPlayground* project's *mute.m* file) that instructs Endpoint Security to mute all events originating from *mds_stores*, a very noisy Spotlight daemon responsible for managing macOS's metadata indexes.

```
❶ #define MDS_STORE "/System/Library/Frameworks/CoreServices.framework/Versions/
A/Frameworks/Metadata.framework/Versions/A/Support/mds_stores"

❷ es_mute_path(client, MDS_STORE, ES_MUTE_PATH_TYPE_LITERAL);
```

Listing 9-1: Muting events from the Spotlight service

After defining the path to the *mds_store* binary ❶, we invoke the `es_mute_path` API ❷, passing it an endpoint client (created previously via a call to `es_new_client`), the path to the *mds_stores* binary, and the `ES_MUTE_PATH_TYPE_LITERAL` enumeration value.

If you instead (or also) want to mute the targets of the events (for example, in a file monitor, the paths to files being created or deleted), use either `ES_MUTE_PATH_TYPE_TARGET_PREFIX` or `ES_MUTE_PATH_TYPE_TARGET_LITERAL`. For instance, if we wanted a file monitor to mute all file events involving the temporary directory associated with the user context under which the monitor process is running, we'd use the code in Listing 9-2.

```
❶ char tmpDirectory[PATH_MAX] = {0};
  realpath([NSTemporaryDirectory() UTF8String], tmpDirectory);

❷ es_mute_path(client, tmpDirectory, ES_MUTE_PATH_TYPE_TARGET_PREFIX);
```

Listing 9-2: Muting all events in the current user's temporary directory

We retrieve the temporary directory with the `NSTemporaryDirectory` function and then resolve any symbolic links in this path (for example, resolving */var* to */private/var*) with the `realpath` function ❶. Next, we mute all file I/O events whose target paths fall within this directory ❷.

Let's compile and run the *ESPlayground* project from the terminal with root privileges. When we launch the Calculator app via Spotlight, it should print out various Endpoint Security events, such as file open and close events:

```
# ESPlayground.app/Contents/MacOS/ESPlayground -mute

ES Playground
Executing 'mute' logic

muted process: /System/Library/Frameworks/
CoreServices.framework/Versions/A/Frameworks/Metadata.framework/Versions/A/Support/mds_stores

muted directory: /private/var/folders/zz/zyxvpxvq6csfxvn_n0000000000000/T

event: ES_EVENT_TYPE_NOTIFY_OPEN
process: /System/Library/CoreServices/Spotlight.app/Contents/MacOS/Spotlight
file path: /System/Applications/Calculator.app/Contents/MacOS/Calculator

event: ES_EVENT_TYPE_NOTIFY_CLOSE
process: /System/Library/CoreServices/Spotlight.app/Contents/MacOS/Spotlight
file path: /System/Applications/Calculator.app/Contents/MacOS/Calculator

event: ES_EVENT_TYPE_NOTIFY_OPEN
process: /System/Applications/Calculator.app/Contents/MacOS/Calculator
file path: /
```

But because we specified the `-mute` flag, we won't receive any events originating from the *mds_stores* daemon or from within the root user's temporary directory. We can confirm this fact by simultaneously running

a file monitor that implements no muting. Notice that this time, we receive such events:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_OPEN",
  "file" : {
    "destination" : "/private/var/folders/zz/zyxvpxvq6csfxvn_n0000000000000/T",
    "process" : {
      "pid" : 540,
      "name" : "mds_stores",
      "path" : "/System/Library/Frameworks/CoreServices.framework/
      Versions/A/Frameworks/Metadata.framework/Versions/A/Support/mds_stores"
    }
  }
  ...
}
```

Endpoint Security has several other muting-related APIs worth mentioning. The `es_mute_process` function provides another way to mute events from a specific process:

```
es_return_t
es_mute_process(es_client_t* _Nonnull client, const audit_token_t* _Nonnull audit_token);
```

As the definition shows, the function expects a client and an audit token of the process to mute. Because it takes an audit token instead of a path (as with the `es_mute_path` function), you can mute a specific instance of a running process. For example, you most likely want to mute events that originate from your own Endpoint Security tool. Using the `getAuditToken` function covered in Chapter 1, Listing 9-3 performs such a muting.

```
NSData* auditToken = getAuditToken(getpid());

es_mute_process(client, auditToken.bytes);
```

Listing 9-3: An ES client muting itself

Besides muting a process entirely, you can also mute just a subset of its events via the `es_mute_process_events` API:

```
es_return_t es_mute_process_events(es_client_t* _Nonnull client, const audit_token_t*
_Nonnull audit_token, const es_event_type_t* _Nonnull events, size_t event_count);
```

After passing a client and an audit token of the process whose events you intend to mute, you should pass an array of events containing the events to mute, as well as the size of the array.

For each muting API, you'll find a corresponding unmuting function, such as `es_unmute_path` and `es_unmute_process`. Moreover, Endpoint Security provides several global unmuting functions. For example, `es_unmute_all_paths` unmutes all muted paths. You can find more details about these functions in Apple's Endpoint Security developer documentation.[1]

Mute Inversion

Mute inversion, a capability added to Endpoint Security in macOS 13, inverts the logic for for muting, both for processes triggering the events and the events themselves. This allows you, for example, to subscribe to events for a very specific set of processes, directories, or files. You'll find it useful for tasks such as the following:

- Detecting unauthorized access to user directories, perhaps by ransomware attempting to encrypt user files or stealers attempting to access authentication tokens or cookies[2]
- Implementing tamper-resistant mechanisms to protect your security tool[3]
- Capturing events triggered by the actions of a malware specimen during analysis or profiling

For example, consider MacStealer, a malware specimen that goes after user cookies.[4] If we decompile its compiled Python code, we can see that it contains a list of common browsers, such as Chrome and Brave, as well as logic to extract their cookies:

```
class Browsers:
def __init__(self, decrypter: object) -> object:
    ...
    self.cookies_path = []
    self.extension_path = []
    ...
    self.cookies = []
    self.decryption_keys = decrypter
    self.appdata = '/Users/*/Library/Application Support'
    self.browsers = {...
        'google-chrome':self.appdata + '/Google/Chrome/',
        ...
        'brave':self.appdata + '/BraveSoftware/Brave-Browser/',
        ...
    }
    ...
def browser_db(self, data, content_type):
    ...
    else:
        if content_type == 'cookies':
           sql = 'select name,encrypted_value,host_key,path,is_secure,..., from cookies'
           keys = ['name', 'encrypted_value', 'host_key', 'path', ..., 'expires_utc']
    ...
    if __name__ == '__main__':
        decrypted = {}
        browsers = Browsers()
        paths = browsers.browser_data()
```

The code exfiltrates the collected cookies, giving the malware authors access to a user's logged-in accounts. By leveraging mute inversion, we can subscribe to file events covering the locations of browser cookies. Any process that attempts to access browser cookies will trigger these events,

including MacStealer, providing a mechanism to detect and thwart its unauthorized actions.

Beginning Mute Inversion

To invert muting, invoke the `es_invert_muting` function, which takes an Endpoint Security client as well as the mute inversion type:

```
es_return_t es_invert_muting(es_client_t* _Nonnull client, es_mute_inversion_type_t mute_type);
```

You can find the mute inversion types in the *ESTypes.h* header file:

```
typedef enum {
    ES_MUTE_INVERSION_TYPE_PROCESS,
    ES_MUTE_INVERSION_TYPE_PATH,
    ES_MUTE_INVERSION_TYPE_TARGET_PATH,
    ES_MUTE_INVERSION_TYPE_LAST
} es_mute_inversion_type_t;
```

The first two types allow you to mute-invert a process. The first type should be used when you're looking to mute-invert a process via its audit token, for example, via the `es_mute_process` API. On the other hand, the second type, `ES_MUTE_INVERSION_TYPE_PATH`, provides the means to identify the process to mute-invert by its path. Finally, `ES_MUTE_INVERSION_TYPE_TARGET_PATH` should be used when instead you're looking to mute-invert events related to the target path, such as a directory.

Mute inversion applies globally across the specified mute inversion type; that is to say, if you invoked `es_invert_muting` with the `ES_MUTE_INVERSION_TYPE_PATH` type, all muted process paths would unmute. For this reason, it often makes sense to create a new Endpoint Security client specifically for mute inversion. (While the system imposes a limit on the number of clients, your program can create at least several dozen of them before causing an `ES_NEW_CLIENT_RESULT_ERR_TOO_MANY_CLIENTS` error.) Also worth nothing is that since muting inversion will only occur for the specified mute inversion type, you can mix and match mute and mute inversions. For example, you could mute processes while mute-inverting paths found in the events. This would be useful in a scenario where you are perhaps building a directory monitor leveraging mute inversion but want to ignore (mute) events from trusted system processes.

Mute inversions also impact the *default mute set*, a handful of paths to system-critical platform binaries that get muted by default. You can invoke the `es_muted_paths_events` function to retrieve a list of all muted paths, including the default ones. The default mute set aims to protect clients from deadlocks and timeout panics, so you likely won't want to generate events for its paths. To avoid doing so, consider invoking `es_unmute_all_paths` before any process-path mute inversions or `es_unmute_all_target_paths` before any target-path mute inversions.

Now that you have inverted muting (for example, via the `es_invert_muting` API), you can invoke any of the corresponding, previously mentioned muting APIs, whose muting logic will now be inverted. This is clearly illustrated

in the next section, which makes use of mute inversion to monitor file access within a single directory.

Monitoring Directory Access

Listing 9-4 is a snippet of mute inversion code that monitors the opening of files in the logged-in user's *Documents* directory. You can find the full implementation in the `muteInvert` function, in the *ESPlayground* project's *muteInvert.m* file.

In "Authorization Events" on page 213, we'll combine this approach with authorization access, a useful protection mechanism that could, for example, block ransomware or malware attempting to access sensitive user files.

```
NSString* consoleUser =
(__bridge_transfer NSString*)SCDynamicStoreCopyConsoleUser(NULL, NULL, NULL); ❶

NSString* docsDirectory =
[NSHomeDirectoryForUser(consoleUser) stringByAppendingPathComponent:@"Documents"];

es_client_t* client = NULL;
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_OPEN};

es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    // Add code here to handle delivered events.
});

es_unmute_all_target_paths(client); ❷
es_invert_muting(client, ES_MUTE_INVERSION_TYPE_TARGET_PATH); ❸
es_mute_path(client, docsDirectory.UTF8String, ES_MUTE_PATH_TYPE_TARGET_PREFIX); ❹

es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Listing 9-4: Monitoring file-open events in the user's Documents *directory*

First, we dynamically build the path to the logged-in user's *Documents* directory. Because Endpoint Security code always runs with root privileges, most APIs that return the current user would simply return the root. Instead, we make use of the `SCDynamicStoreCopyConsoleUser` API to get the name of the user currently logged in to the system ❶. Note that the API isn't aware of the automatic reference counting (ARC) memory management feature, so we add `__bridge_transfer`, which saves us from having to manually free the memory containing the user's name. Next, we invoke the `NSHomeDirectoryForUser` function to get the home directory, to which we then append the path component *Documents*.

After defining the events of interest and creating a new Endpoint Security client, the code unmutes all target paths ❷. Then it invokes `es_invert_muting` with the `ES_MUTE_INVERSION_TYPE_TARGET_PATH` value to invert muting ❸. Next, the code invokes `es_mute_path`, passing in the document's directory ❹. Since we've inverted muting, this API instructs Endpoint Security to deliver only events that occur in this directory and ignore all others. Finally, we invoke `es_subscribe` with the events of interest to commence the delivery of such events.

To complete this example, print out the event, which you'll recall gets delivered to the `es_handler_block_t` callback block specified in the last parameter to the `es_new_client`. Listing 9-5 shows an inline implementation.

```
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
  ❶ es_string_token_t* procPath = &message->process->executable->path;
  ❷ es_string_token_t* filePath = &message->event.open.file->path;

  ❸ printf("event: ES_EVENT_TYPE_NOTIFY_OPEN\n");
    printf("process: %.*s\n", (int)procPath->length, procPath->data);
    printf("file path: %.*s\n", (int)filePath->length, filePath->data);
});
```

Listing 9-5: Printing out a file-open Endpoint Security event

We extract the path to the responsible process. We can always find this process in the `message` structure passed by reference to the handler block. To get its path, we check the `process` structure's `executable` member ❶. Next, we extract the path of the file that the process has attempted to open. For `ES_EVENT_TYPE_NOTIFY_OPEN` events, we find this path in an `es_event_open_t` structure, located in the `message` structure's `event` member ❷. After extracting the paths for the responsible process and file, we print them out ❸.

The tool should now detect any access to files in the *Documents* directory. You can test this by running *ESPlayground* with the `-muteinvert` flag. You'll see that it displays no Endpoint Security events unless they originate within *Documents*. You can trigger such events by either browsing to the directory via Finder or using the terminal (for example, to list the directory's contents via `ls`):

```
# ESPlayground.app/Contents/MacOS/ESPlayground -muteinvert

ES Playground
Executing 'mute inversion' logic
unmuted all (default) paths
mute (inverted) /Users/Patrick/Documents

event: ES_EVENT_TYPE_NOTIFY_OPEN
process: /System/Library/CoreServices/Finder.app/Contents/MacOS/Finder
file path: /Users/Patrick/Documents

event: ES_EVENT_TYPE_NOTIFY_OPEN
process: /bin/ls
file path: /Users/Patrick/Documents
```

If we extended the example code to also monitor other directories, such as those where browsers store their cookies, we'd easily detect stealers such as MacStealer! In the next section, I'll cover the powerful authorization event type.

Authorization Events

Unlike notification-based events, which an Endpoint Security client receives after some activity occurs on the system, authorization events allow a client to examine and then allow or deny events *before* they've completed. This feature provides a mechanism for building security tools capable of proactively detecting and thwarting malicious activity. Although working with authorization events involves similar concepts as working with notification events, there are some important differences. To explore these, let's dive into the code.

Conceptually, our goal is simple: design a tool capable of blocking the execution of non-notarized programs originating from the internet. As we've seen, the overwhelming majority of macOS malware isn't notarized, while legitimate software almost always is, making this a powerful approach to stopping malware. When a user attempts to launch an item downloaded from the internet, we'll intercept this execution before it's allowed, then check its notarization status. We'll allow validly notarized items and block all others.

At the time of this writing, recent versions of macOS attempt to implement this same check, but they do so less rigorously. First, up until macOS 15, if the user right-clicks a download item, the operating system still provides the option to run non-notarized items. Malware authors are, of course, well aware of this loophole and often leverage it to get their untrusted malware to execute. The prolific macOS adware Shlayer and many macOS stealers are fond of this trick. Moreover, Apple's implementation to prevent non-notarized code on macOS has been rife with exploitable bugs (such as CVE-2021-30657 and CVE-2021-30853), rendering it essentially useless.[5]

I implemented a notarization check in one of Objective-See's most popular tools, BlockBlock, discussed in detail in Chapter 11. When run in notarization mode, this tool blocks any downloaded binary that isn't notarized, including malware that attempts to exploit CVE-2021-30657 and CVE-2021-30853, well before patches from Apple were available.[6] We'll roughly follow BlockBlock's approach here. Note that in your own implementation, you might take a less draconian approach; for example, rather than blocking all non-notarized items, you might block only those that users may have been tricked into running. (In macOS 15, Apple introduced the `ES_EVENT_TYPE_NOTIFY_GATEKEEPER_USER_OVERRIDE` event you may be able to leverage to detect this.) Or you might collect non-notarized binaries for external analysis or subject them to other heuristics mentioned in this book before deciding whether to prevent their execution.

Creating a Client and Subscribing to Events

In this section, we subscribe to Endpoint Security authorization events before discussing how to respond to such events in a timely manner. You can find a full implementation of the code mentioned in this section in the `authorization` function, found in the *ESPlayground* project's *authorization.m* file.

As when working with notification events, we start by creating an Endpoint Security client, specify an `es_handler_block_t` block, and subscribe to events of interest (Listing 9-6).

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

es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    // Add logic to allow or block processes.
});

es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Listing 9-6: Subscribing to authorization events for process executions

To block non-notarized processes, we need to subscribe to only a single authorization event: `ES_EVENT_TYPE_AUTH_EXEC` ❶. Apple's developer documentation succinctly describes it as the event type for any process that "requests permission from the operating system to execute another image."[7] Once the call to `es_subscribe` returns, Endpoint Security will invoke our code anytime a new process is about to be executed.

Next, we must respond to the operating system with a decision to either authorize or deny the delivered event. To respond, we use the `es_respond_auth_result` API, defined as follows in *ESClient.h*:

```
es_respond_result_t es_respond_auth_result(es_client_t* _Nonnull client,
const es_message_t* _Nonnull message, es_auth_result_t result, bool cache);
```

The function takes the client that received the message, the delivered message, the authorization result, and a flag indicating whether the results should be cached. To allow a message, invoke this function with an `es_auth_result_t` value of `ES_AUTH_RESULT_ALLOW`. To deny the message, specify a value of `ES_AUTH_RESULT_DENY`. If you pass in `true` for the cache flag, Endpoint Security will cache the authorization decision, meaning future events from the same process may not trigger additional authorization events. This, of course, has performance benefits, though some important nuances to be aware of. First, imagine that you've cached an authorization decision for a process execution event. Even if that process is executed with different arguments, no additional authorization event will be generated, which could be problematic if a detection heuristic makes use of process arguments. Second, be aware that the cache is global for the system, meaning if any other Endpoint Security client does not cache an event, you'll still receive it (even if you've previously cached it).

Let's build upon the code in Listing 9-6 to extract the path of the process about to be spawned and then determine how to respond. For simplicity, we'll just allow all processes in this example (Listing 9-7).

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

es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
  ❶ es_process_t* process = message->event.exec.target;
  ❷ es_string_token_t* procPath = &process->executable->path;
```

```
        printf("\nevent: ES_EVENT_TYPE_AUTH_EXEC\n");
        printf("process: %.*s\n", (int)procPath->length, procPath->data);

      ❸ es_respond_auth_result(client, message, ES_AUTH_RESULT_ALLOW, false);
    });

    es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Listing 9-7: Handling process authorization events

Within the callback block, we extract information about the process that is about to be spawned. First, we get a pointer to its `es_process_t` structure, found with the `es_event_exec_t` structure in the Endpoint Security message ❶. From this, we extract just its path ❷ and print it out. Finally, we invoke the `es_respond_auth_result` API with `ES_AUTH_RESULT_ALLOW` to tell the Endpoint Security subsystem to authorize that process's execution ❸.

NOTE *In* ESTypes.h*, Apple specifies an important but easy-to-overlook nuance: for file authorization events (`ES_EVENT_TYPE_AUTH_OPEN`) only, your code must provide an authorization response via the `es_respond_flags_result` function, not via the `es_respond_auth_result` function. The same header file notes that when invoking the `es_respond_flags_result` function, you should pass a value of `0` to deny the event and `UINT32_MAX` to allow it.*

Let's run *ESPlayground* with the `-authorization` flag and then launch the Calculator application:

```
# ESPlayground.app/Contents/MacOS/ESPlayground -authorization

ES Playground
Executing 'authorization' logic

event: ES_EVENT_TYPE_AUTH_EXEC
process: /System/Applications/Calculator.app/Contents/MacOS/Calculator
```

We see the authorization event, and because we're allowing all processes, Endpoint Security doesn't block it.

Meeting Message Deadlines

There is one very important caveat to responding to authorization events: if we miss the response deadline, Endpoint Security will allow the event and forcefully kill our client.

```
Exception Type:      EXC_CRASH (SIGKILL)
Exception Codes:     0x0000000000000000, 0x0000000000000000
Termination Reason:  Namespace ENDPOINTSECURITY, Code 2 EndpointSecurity client
terminated because it failed to respond to a message before its deadline
```

From a system and usability point of view, this approach makes sense. If the program takes too long to respond, the entire system could lag or, worse, hang.

The `es_message_t` structure has a field named `deadline` that tells us exactly how long we have to respond to the message. The header file also notes that the deadline can vary substantially between each message; thus, our code should inspect each message's deadline accordingly.

Let's look at how BlockBlock's process monitoring logic handles deadlines.[8] Deadlines are especially important for this tool, as it waits for the user's input before authorizing or denying the non-notarized process, meaning it faces a very real possibility of hitting the deadline (Listing 9-8).

```
❶ dispatch_semaphore_t semaphore = dispatch_semaphore_create(0);
❷ uint64_t deadline = message->deadline - mach_absolute_time();

❸ dispatch_async(dispatch_get_global_queue(QOS_CLASS_DEFAULT, 0), ^{
    ❹ if(0 != dispatch_semaphore_wait(semaphore,
      dispatch_time(DISPATCH_TIME_NOW, machTimeToNanoseconds(deadline)
      - (1 * NSEC_PER_SEC)))) {
        ❺ es_respond_auth_result(client, message, ES_AUTH_RESULT_ALLOW, false);
    }
});
```

Listing 9-8: BlockBlock's handling of Endpoint Security message deadlines

First, the code creates a semaphore ❶ and computes the deadline ❷. Because Endpoint Security reports the message deadline in absolute time, the code subtracts the current time from it to figure out how long it has left. Next, the code submits a block to execute asynchronously in a background queue ❸, where it delivers the message to the user and, in another asynchronous block, waits for the response. I've omitted this part of the code to keep things concise, as its specifics aren't relevant.

Performing time-consuming processing in another asynchronous queue allows the code to signal the semaphore once the processing is complete and avoid the timeout, which the code sets up next ❹. Once BlockBlock has delivered the message to the user and is awaiting a response, it invokes the `dispatch_semaphore_wait` function to wait on the semaphore until a certain time. You probably guessed it: the function waits until right before the message's deadline is hit. If a timeout occurs (meaning a user response didn't signal the semaphore and the message deadline is about to be hit), the code has no choice but to respond, which it does by defaulting to authorizing the event ❺.

Note that the Mach absolute time value returned by a function can vary between processes, depending on whether they're native or translated. To maintain consistency, you should apply a timebase, which you can retrieve using the `mach_timebase_info` function. Apple documentation illustrates this in the following code, which converts a mach time value to nanoseconds using timebase information:

```
uint64_t MachTimeToNanoseconds(uint64_t machTime) {
    uint64_t nanoseconds = 0;
    static mach_timebase_info_data_t sTimebase;
    if (sTimebase.denom == 0)
        (void)mach_timebase_info(&sTimebase);

    nanoseconds = ((machTime * sTimebase.numer) / sTimebase.denom);
    return nanoseconds;
}
```

You might have noticed that the code in Listing 9-8 leveraged this function when computing the wait time for the dispatch semaphore.

NOTE *If you're asynchronously processing Endpoint Security messages, such as when asking a user for input and awaiting their response, you must retain the message via the `es_retain_message` API. Once you're done with the message, you must release it with a call to `es_release_message`.*

Now that you've seen how to respond to Endpoint Security authorization events while taking deadlines into account, you're ready to look at the last piece of the "blocking non-notarized processes" puzzle.

Checking Binary Origins

Once we've registered for `ES_EVENT_TYPE_AUTH_EXEC` events, the system will invoke the `es_handler_block_t` block passed to the `es_new_client` function before each new process is spawned. In this block, we'll add logic to deny non-notarized processes from remote locations only. That last part is important, as local platform binaries aren't notarized but should, of course, be allowed. Along the same lines, you may want to consider allowing applications from the official Mac App Store. Though not notarized, they've passed a similar and (hopefully) stringent Apple review process.

To determine if a process's binary originated from a remote location, we'll defer to macOS by checking whether the binary has been translocated or has the `com.apple.quarantine` extended attribute. If either condition is true, the operating system has marked the item as originating from a remote source. *Translocation* is a security mitigation built into recent versions of macOS designed to thwart relative dynamic library hijacking attacks.[9]

In short, when a user attempts to open an executable item from a downloaded disk image or ZIP file, macOS will first create a random read-only mount containing a copy of the item, then launch this copy. If we can programmatically determine that a process about to be executed has been translocated, we know we should subject it to a notarization check.

To check if an item has been translocated, we can invoke the private `SecTranslocateIsTranslocatedURL` API. This function takes several parameters, including the path of the item to check and a pointer to a Boolean flag that macOS will set to true if it has translocated the item. Because the API is private, we must dynamically resolve it before we can invoke it. The code in Listing 9-9 does both tasks.[10]

```
#import <dlfcn.h>
BOOL isTranslocated(NSString* path) {
    BOOL isTranslocated = NO;
    void* handle = dlopen(
    "/System/Library/Frameworks/Security.framework/Security", RTLD_LAZY); ❶

    BOOL (*SecTranslocateIsTranslocatedURL)(CFURLRef path, bool* isTranslocated,
    CFErrorRef* __nullable error) = dlsym(handle,"SecTranslocateIsTranslocatedURL"); ❷

    SecTranslocateIsTranslocatedURL((__bridge CFURLRef)([NSURL fileURLWithPath:path]),
    &isTranslocated, NULL); ❸

    return isTranslocated;
}
```

Listing 9-9: A helper function that uses private APIs to determine whether an item has been translocated

The code loads the *Security* framework, which contains the `SecTranslocateIsTranslocatedURL` API ❶. Once it's loaded, the code resolves the API via `dlsym` ❷, then invokes the function with the path of the item to check ❸. When the API returns, it will set the second parameter to the result of the translocation check.

Another way to check whether an item has a remote origin is via the `com.apple.quarantine` extended attribute, added either by the application responsible for downloading the item or by the operating system directly, if the application has set `LSFileQuarantineEnabled = 1` in its *Info.plist* file. You can programmatically retrieve the value of an item's extended attribute using various private `qtn_file_*` APIs found in */usr/lib/system/libquarantine.dylib*, though you must first dynamically resolve these functions. Invoke them in the following manner:

1. Invoke `qtn_file_alloc` to allocate a `_qtn_file` structure.
2. Invoke the `qtn_file_init_with_path` API with the `_qtn_file` pointer and the path of the item whose quarantine attributes you wish to retrieve. If this function returns `QTN_NOT_QUARANTINED` (`-1`), the item isn't quarantined.
3. Invoke the `qtn_file_get_flags` API with the `_qtn_file` pointer to retrieve the actual value of the `com.apple.quarantine` extended attribute.
4. If the `qtn_file_init_with_path` function didn't return `QTN_NOT_QUARANTINED`, you'll know that the item is quarantined, but you may want to check whether a user previously approved the file. You can determine this by checking the value returned by `qtn_file_get_flags`, where the `QTN_FLAG_USER_APPROVED` (0x0040) bit may be set.
5. Make sure to free the `_qtn_file` structure by calling `qtn_file_free`.

In several cases, macOS didn't appropriately classify nonlocal items as having originated from a remote source. For example, in CVE-2023-27951, the operating system failed to apply the `com.apple.quarantine` extended attribute. In production code, you might therefore want to take a more comprehensive approach to determining a binary's origins. For instance,

you could create a file monitor to detect binary downloads and then subject these binaries to the notarization checks, or just block any nonplatform binary that isn't notarized. And, yes, malware (once it's off and running) may remove the quarantine extended attribute from other components it has downloaded prior to their execution to potentially bypass macOS or BlockBlock checks. As such, you may also want to subscribe to the `ES_EVENT_TYPE_AUTH_DELETEEXTATTR` Endpoint Security event, which will be able to detect and prevent the removal of the quarantine attribute.

Now that we can determine whether a process originated from a remote source, we must check whether the binary backing the process is notarized. As you saw in Chapter 1, this is as easy as invoking the `SecStaticCodeCheckValidity` API with the appropriate requirement string.

If BlockBlock ascertains that the process about to be executed is from a remote source and not notarized, it will alert the user to request their input. If the user decides that the process is, for example, untrustworthy or unrecognized, BlockBlock will invoke the function in Listing 9-10 to block it.

```
-(BOOL)block:(Event*)event {
    BOOL blocked = NO;

    if(YES != (blocked = [self respond:event action:ES_AUTH_RESULT_DENY])) {
        os_log_error(logHandle, "ERROR: failed to block %{public}@", event.process.name);
    }

    return blocked;
}
```

Listing 9-10: Blocking untrustworthy processes

It invokes the `respond:action:` method with the `ES_AUTH_RESULT_DENY` constant. If we look at this method, we see that, at its core, it just invokes `es_respond_auth_result`, passing along the specified allow or deny action to the Endpoint Security subsystem. Also, as `true` is passed in for the cache flag, subsequent executions of the same process will not generate additional authorization events, thus providing a noticeable performance boost (Listing 9-11).

```
-(BOOL)respond:(Event*)event action:(es_auth_result_t)action {
    ...
    result = es_respond_auth_result(event.esClient, event.esMessage, action, true);
    ...
}
```

Listing 9-11: Passing Endpoint Security the action to take

For a full implementation that blocks non-notarized processes via Endpoint Security, see BlockBlock's process plug-in.[11]

Blocking Background Task Management Bypasses

Let's consider another example that uses Endpoint Security authorization events to detect malware, this time by focusing on attempts to leverage

exploits that bypass built-in macOS security mechanisms. While the use of these exploits isn't yet widespread, the inclusion of new security mechanisms in macOS has increasingly forced malware to employ new techniques to achieve their malicious objectives, so monitoring for these exploits may aid your detections.

In Chapter 5, I discussed macOS's new Background Task Management (BTM) database, which monitors for persistent items, generates alerts for them, and globally tracks their behavior. BTM is problematic for malware hoping to persist, because users will now receive an alert when the malware gets installed. For example, Figure 9-1 shows the BTM alert that users receive when malware known as DazzleSpy persistently installs itself as a binary named *softwareupdate*.

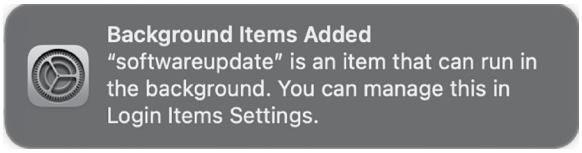

Figure 9-1: A BTM alert showing that a binary named softwareupdate *has been persistently installed*

Luckily for the malware, my research into BTM revealed that Apple's original implementation was easy to subvert in several ways, preventing this alert. This section details two such bypasses and shows how to leverage Endpoint Security to detect and block these subversions. Note that I informed Apple about these issues, and, at least in macOS 15 (and perhaps on earlier versions of macOS), they appear to have been fixed. Even so, you could adapt the code in this section to detect other local exploits.

Manual Database Resets

The first method of bypassing BTM was incredibly simple. Recall that Chapter 5 discussed `sfltool`, which ships with macOS and allows users to interface with the BTM database. One of its command line options, `resetbtm`, will clear the database, causing it to be rebuilt. Once this command is run, however, the system won't deliver subsequent BTM alerts until it reboots, even though items can still persist.

Thus, malware wanting to avoid generating BTM alerts could simply execute `sfltool` with the `resetbtm` command before executing its persistence code. The technique has yet to be observed in the wild but is easy to exploit, as shown in the following log message, generated after a manual database reset. These message shows that while the BTM daemon detected DazzleSpy's persistent install, it decided not to post an advisory alert:

```
% log stream
backgroundtaskmanagementd: registerLaunchItem: result=no error, new item
disposition=[enabled, allowed, visible, not notified],
identifier=com.apple.softwareupdate,
```

```
url=file:///Users/User/Library/LaunchAgents/com.apple.softwareupdate.plist
backgroundtaskmanagementd: should post advisory=false for uid=501, id=
6ED3BEBC-8D60-45ED-8BCC-E0163A8AA806, item=softwareupdate
```

Under normal circumstances, users have no reason to reset the BTM database. So, we can thwart this exploit by subscribing to Endpoint Security process events and blocking the spawning of `sfltool` when it is executed with the `resetbtm` argument.

To detect the execution of processes, including `sfltool`, we can register for the `ES_EVENT_TYPE_NOTIFY_EXEC` event discussed in Chapter 8. We can access the process's path via the `es_process_t` process structure and extract its arguments with the `es_exec_arg_count` and `es_exec_arg` helper functions. Once you've extracted the path and arguments, simple string comparisons should tell you if the reported process event is a result of `sfltool` spawned with the `resetbtm` argument.

Of course, you'll likely want to block these events, which you can do by registering for `ES_EVENT_TYPE_AUTH_EXEC`. This event's callback will be invoked with an Endpoint Security message containing a pointer to an `es_process_t` structure. From this, you can extract both the path and the arguments of the process about to be spawned, then block the spawning by invoking the `es_respond_auth_result` function with a value of `ES_AUTH_RESULT_DENY`.

Stop Signals

While researching the BTM subsystem, I came across another trivial way to bypass its alerts.[12] In short, malware could easily send a stop (`SIGSTOP`) signal to the BTM agent responsible for displaying the persistence advisory message to the user. Once this component halted, the malware could persist without the user being alerted. To detect and block this bypass, we can lean on Endpoint Security once again. As it's extremely unlikely that a user would send a `SIGSTOP` signal to the BTM agent under normal circumstances, we can assume this event is malware attempting to subset the subsystem.

The year following my presentation, researchers at Sentinel One uncovered malware taking a similar (albeit less elegant) approach. In their report,[13] the researchers noted that the malicious code would continually send a kill signal to macOS's Notification Center process to block the BTM's persistence advisory message, which the system would normally display when the malware persisted.

We can detect signals with the `ES_EVENT_TYPE_NOTIFY_SIGNAL` event or, better yet, block signals altogether with the corresponding authorization event, `ES_EVENT_TYPE_AUTH_SIGNAL`. In Listing 9-12, we focus on the latter task.

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

es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    int signal = message->event.signal.sig; ❶
    es_process_t* sourceProcess = message->process; ❷
    es_process_t* targetProcess = message->event.signal.target; ❸
```

```
    // Add code to check if signal is a SIGSTOP or SIGKILL being sent to a process
    // involved in showing user notification alerts.

});

es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Listing 9-12: Subscribing to authorization events for signal deliveries

Whenever a process attempts to send a signal, Endpoint Security will invoke the callback with a message containing an `es_event_signal_t` structure. The code extracts the type of signal ❶, as well as the source ❷ and target processes ❸.

We can check whether the signal is a `SIGSTOP` or `SIGKILL` and whether the process that would receive the signal is either the BTM agent or the Notification Center. If so, we simply deny the signal delivery by invoking `es_respond_auth_result` with the `ES_AUTH_RESULT_DENY` value (Listing 9-13).

```
if( (signal == SIGSTOP) || (signal == SIGKILL) ) {
    pid_t targetPID = audit_token_to_pid(targetProcess->audit_token);

    if( (targetPID == btmAgentPID) || (targetPID == notificationCenterPID) ) {
        es_respond_auth_result(client, message, ES_AUTH_RESULT_DENY, false);
    }
}
```

Listing 9-13: Denying suspicious `SIGSTOP` or `SIGKILL` signals

Note that elsewhere in your code, you should probably look up and save the process ID for the BTM agent and Notification Center process, as you wouldn't want to look it up each time a signal is delivered. You'd also likely want to log a message that includes information about the source process attempting to send the suspicious signal or else collect it for further examination.

If you implement this code, compile it, run it, and then manually attempt to subvert the notifications from the BTM subsystem by stopping the agent, your actions should now fail:

```
% pgrep BackgroundTaskManagementAgent
590

% kill -SIGSTOP 590
kill: kill 590 failed: operation not permitted
```

In the terminal, we get the process ID of the BTM agent (590, in this instance). Then we use the `kill` command to send a `SIGSTOP` signal to the agent. This will trigger the delivery of an `ES_EVENT_TYPE_AUTH_SIGNAL` event to our program, which will deny it, resulting in the "operation not permitted" message.

Building a File Protector

I'll wrap up the discussion of the Endpoint Security framework by developing a proof-of-concept file protector. You can find its full implementation in the `protect` function, in the *ESPlayground* project's *protect.m* file.

Our code will monitor a specific directory (for example, the user's home directory or the directory containing browser cookies) and allow only authorized processes to access it. Whenever a process attempts to access a file in the directory, Endpoint Security will trigger an authorization event, giving our code an opportunity to closely examine the process and decide whether to allow it. In this example, we'll allow only platform and notarized binaries and block the rest.

This file protector is conceptually similar to Apple's Transparency, Consent, and Control (TCC), but it adds another level of protection. After all, users may naively grant TCC permissions to malware, making previously protected files accessible, and malware often exploits or bypasses TCC itself, as in the case of the XCSSET malware.[14] Finally, you may want to provide authorized access (and detect unauthorized access) to files located outside TCC's protected directories, such as the cookies files for certain third-party browsers.

Earlier in this chapter, I discussed monitoring the logged-in user's *Documents* directory via a notify event. The code in this section is similar, except it covers the user's entire home directory and extends the list of events of interest to also include those related to attempted file deletions. Most notably, this code leverages Endpoint Security authorization events to proactively block untrusted access. As usual, we'll start by specifying the Endpoint Security events of interest, creating an Endpoint Security client, setting up muting inversion, and finally subscribing to the events (Listing 9-14).

```
NSString* consoleUser =
(__bridge_transfer NSString*)SCDynamicStoreCopyConsoleUser(NULL, NULL, NULL);

NSString* homeDirectory = NSHomeDirectoryForUser(consoleUser);

es_client_t* client = NULL;
es_event_type_t events[] = {ES_EVENT_TYPE_AUTH_OPEN, ES_EVENT_TYPE_AUTH_UNLINK}; ❶

es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    // Add code here to implement logic to examine process and respond to event.
});

es_unmute_all_target_paths(client); ❷
es_invert_muting(client, ES_MUTE_INVERSION_TYPE_TARGET_PATH);
es_mute_path(client, homeDirectory.UTF8String, ES_MUTE_PATH_TYPE_TARGET_PREFIX); ❸

es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Listing 9-14: Setting up an Endpoint Security client to authorize file access

Several Endpoint Security authorization events relate to file access. Here, we use `ES_EVENT_TYPE_AUTH_OPEN` and `ES_EVENT_TYPE_AUTH_UNLINK` ❶, which give us the ability to authorize programs that attempt to open or delete files. The former event can detect a range of malware with either ransomware or stealer capabilities, while the latter event could perhaps detect and prevent malware with wiper capabilities that might try to delete or wipe important files.

After creating a new Endpoint Security client (whose handler block we'll write shortly) ❷, the code sets up muting inversion ❸, given that we're interested only in events related to the directory we're about to specify. It dynamically builds a path to the logged-in user's home directory, then invokes the `es_mute_path` API. Because we've inverted muting, this API tells the Endpoint Security subsystem to deliver events that occur within the specified path only. After the code calls `es_subscribe`, Endpoint Security will start delivering events by executing the handler block specified in the call to the `es_new_client` function.

How might we implement such a block? To keep things simple, let's first assume we'll allow any access (Listing 9-15).

```
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    switch(message->event_type) {
        case ES_EVENT_TYPE_AUTH_OPEN:
            es_respond_flags_result(client, message, UINT32_MAX, false); ❶
            break;
        case ES_EVENT_TYPE_AUTH_UNLINK:
            es_respond_auth_result(client, message, ES_AUTH_RESULT_ALLOW, false); ❷
            break;
        ...
    }
});
```

Listing 9-15: Allowing all file accesses

Recall that for `ES_EVENT_TYPE_AUTH_OPEN` events, Apple documentation states that we have to respond with the `es_respond_flags_result` function ❶. To tell the Endpoint Security subsystem to allow the event, we invoke this function with `UINT32_MAX`. For the `ES_EVENT_TYPE_AUTH_UNLINK` event, we respond using `es_respond_auth_result`, as usual ❷.

On the flip side, Listing 9-16 shows the code to deny all file opens or deletions in the directory.

```
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    switch(message->event_type) {
        case ES_EVENT_TYPE_AUTH_OPEN:
            es_respond_flags_result(client, message, 0, false); ❶
            break;
        case ES_EVENT_TYPE_AUTH_UNLINK:
```

```
            es_respond_auth_result(client, message, ES_AUTH_RESULT_DENY, false); ❷
            break;
        ...
    }
});
```

Listing 9-16: Denying all file accesses

The only changes from the code to allow all events is that we now call the `es_respond_flags_result` function ❶ with `0` as its third parameter and pass `es_respond_auth_result` the value `ES_AUTH_RESULT_DENY` ❷.

Let's expand this code to extract the path of the process responsible for the event, as well as the path of the file the process is trying to open or delete (Listing 9-17).

```
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    es_string_token_t* filePath = NULL;
    es_string_token_t* procPath = &message->process->executable->path; ❶

    switch(message->event_type) {
        case ES_EVENT_TYPE_AUTH_OPEN:
            filePath = &message->event.open.file->path; ❷
            es_respond_flags_result(client, message, 0, false);
            break;
        case ES_EVENT_TYPE_AUTH_UNLINK:
            filePath = &message->event.unlink.target->path; ❸
            es_respond_auth_result(client, message, ES_AUTH_RESULT_DENY, false);
            break;
        ...
    }
});
```

Listing 9-17: Extracting process paths and filepaths

We can find the responsible process's path in the `process` member of the `message` structure for any Endpoint Security event ❶, but other information is event specific. Thus, we extract the file in the handler for each event type. For `ES_EVENT_TYPE_AUTH_OPEN` events, we find it in an `es_event_open_t` structure ❷, and for `ES_EVENT_TYPE_AUTH_UNLINK` events, it lives in an `es_event_unlink_t` structure ❸.

Now we should allow or deny file openings and deletions based on some rules, depending on what we're attempting to protect. Recall that the MacStealer malware attempts to steal browser cookies. Generally speaking, no third-party process other than the browser should access its cookies. Thus, you may simply want to implement a deny rule with an exception to allow the browser itself. Via the process ID, path, or, better yet, code signing information, it should be easy to identify whether the browser is the responsible process.

If you're protecting files in the user's home directory, this kind of "deny all with exceptions" approach would likely impact the usability of the system. Thus, you may want to use heuristics, such as authorizing only notarized applications, those from the App Store, or platform binaries. However, malware

sometimes delegates actions to shell commands, which are platform binaries, so you'll likely want to examine the process hierarchy of the responsible process to make sure it's not being abused in malicious ways.

In this example, we'll keep things simple by allowing only platform or notarized binaries to access the current user's home directory (Listing 9-18).

```
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    es_string_token_t* filePath = NULL;
    es_string_token_t* procPath = &message->process->executable->path;

    BOOL isTrusted = ( (YES == message->process->is_platform_binary) ||
    (YES == isNotarized(message->process)) );

    switch(message->event_type) {
        case ES_EVENT_TYPE_AUTH_OPEN:
            filePath = &message->event.open.file->path;
            printf("\nevent: ES_EVENT_TYPE_AUTH_OPEN\n");
            printf("responsible process: %.*s\n", (int)procPath->length, procPath->data);
            printf("target file path: %.*s\n", (int)filePath->length, filePath->data);
            if(YES == isTrusted) {
                printf("process is trusted, so will allow event\n");
                es_respond_flags_result(client, message, UINT32_MAX, false);
            } else {
                printf("process is *not* trusted, so will deny event\n");
                es_respond_flags_result(client, message, 0, false);
            }
            break;

        case ES_EVENT_TYPE_AUTH_UNLINK:
            filePath = &message->event.unlink.target->path;
            printf("\nevent: ES_EVENT_TYPE_AUTH_UNLINK\n");
            printf("responsible process: %.*s\n", (int)procPath->length, procPath->data);
            printf("target file path: %.*s\n", (int)filePath->length, filePath->data);
            if(YES == isTrusted) {
                printf("process is trusted, so will allow event\n");
                es_respond_auth_result(client, message, ES_AUTH_RESULT_ALLOW, false);
            } else {
                printf("process is *not* trusted, so will deny event\n");
                es_respond_auth_result(client, message, ES_AUTH_RESULT_DENY, false);
            }
            break;
        ...
    }
});
```

Listing 9-18: Granting file access for platform and notarized processes only

We check whether the responsible process either is a platform binary or has been notarized. Checking whether a process is a platform binary is as easy as checking the `is_platform_binary` member of the `process` structure found in the delivered Endpoint Security message. In Chapter 3, we used Apple's code signing APIs to figure out whether a process is notarized; we won't cover

this process again here, except to note that we've created a simple helper function named `isNotarized` that uses the responsible process's audit token to check its notarization status. (If you're interested in seeing the full implementation of this function, see the *protect.m* file in the *ESPlayground* project.)

It's also worth pointing out that the logical OR operator will short-circuit if the first condition is true, so we put the platform binary check first. Because it's a simple check against a Boolean value in a structure, it's less computationally intensive than a full notarization check, so we perform the more efficient check first and perform the second check only if needed.

Let's compile the *ESPlayground* project and run it with the `-protect` flag to trigger this logic. The tool detects the use of built-in macOS commands to examine the home directory and delete a file within the *Documents* directory but still allows the actions:

```
# ESPlayground.app/Contents/MacOS/ESPlayground -protect

ES Playground
Executing 'protect' logic
protecting directory: /Users/Patrick

event: ES_EVENT_TYPE_AUTH_OPEN
responsible process: /bin/ls
target file path: /Users/Patrick
process is trusted, so will allow event

event: ES_EVENT_TYPE_AUTH_UNLINK
responsible process: /bin/rm
target file path: /Users/Patrick/Documents/deleteMe.doc
process is trusted, so will allow event
```

Now consider WindTail, a persistent cyber-espionage implant that seeks to enumerate and exfiltrate files in the user's *Documents* directory. If we install it in a virtual machine, we can see the malware (called *Final_Presentation.app*) attempts to enumerate the files in the user's documents directory. We detect this access, and because WindTail's binary (called *usrnode* in this example) isn't trusted, we block access to the directory:

```
# ESPlayground.app/Contents/MacOS/ESPlayground -protect

ES Playground
Executing 'protect' logic
protecting directory: /Users/User

event: ES_EVENT_TYPE_AUTH_OPEN
responsible process: /Users/User/Library/Final_Presentation.app/Contents/MacOS/usrnode
target file path: /Users/User/Documents
process is *not* trusted, so will deny event
```

It's hard to overstate the importance of Endpoint Security for building tools capable of detecting and protecting against Mac malware. In recent

years, Apple has added more events (such as `ES_EVENT_TYPE_NOTIFY_XP_MALWARE_DETECTED` in macOS 13 and `ES_EVENT_TYPE_NOTIFY_GATEKEEPER_USER_OVERRIDE` in macOS 15), and powerful capabilities to the framework, so when building any security tool, using Endpoint Security should be your first consideration.

Conclusion

In this chapter, I covered advanced Endpoint Security topics, including muting, inverted muting, and authorization events. The examples showed you how to use these capabilities to build tools capable of detecting malware when it performs unauthorized actions, as well as proactively thwarting the action in the first place.

This chapter wraps up Part II of this book, dedicated to topics of real-time monitoring capabilities. Part III will put together the many topics covered in Parts I and II as we explore the internals of Objective-See's most popular macOS malware detection tools.

Notes

1. See "Client," Apple Developer Documentation, *https://developer.apple.com/documentation/endpointsecurity/client*.
2. Pete Markowsky (@PeteMarkowsky), "A small list of things you can do with this. 1. lockdown access to your SAAS bearer tokens to specific apps . . . ," X, May 2, 2023, *https://x.com/PeteMarkowsky/status/1653453951839109133*.
3. See *https://github.com/google/santa/blob/8a7f1142a87a48a48271c78c94f830d8efe9afa9/Source/santad/EventProviders/SNTEndpointSecurityTamperResistance.mm#L15*.
4. Shilpesh Trivedi, "MacStealer: Unveiling a Newly Identified MacOS-Based Stealer Malware," *Uptycs*, March 24, 2023, *https://www.uptycs.com/blog/macstealer-command-and-control-c2-malware*.
5. You can read more about these notarization bypass flaws in Patrick Wardle, "All Your Macs Are Belong to Us," Objective-See, April 26, 2021, *https://objective-see.org/blog/blog_0x64.html*, and in Patrick Wardle, "Where's the Interpreter!?," Objective-See, December 22, 2021, *https://objective-see.org/blog/blog_0x6A.html*.
6. Objective-See Foundation (@objective_see), "Did you know BlockBlock . . . ," X, March 2, 2022, *https://x.com/objective_see/status/1499172783502204929*.
7. "ES_EVENT_TYPE_AUTH_EXEC," Apple Developer Documentation, *https://developer.apple.com/documentation/endpointsecurity/es_event_type_t/es_event_type_auth_exec*.
8. See *https://github.com/objective-see/BlockBlock*.

9. You can read about such attacks uncovered by yours truly in Patrick Wardle, "Dylib Hijacking on OS X," VirusBulletin, March 19, 2015, *https://www.virusbulletin.com/blog/2015/03/paper-dylib-hijacking-os-x*.

10. The code in Listing 9-9 is inspired by Jeff Johnson, "Detect App Translocation," Lapcat Software, July 26, 2016, *https://lapcatsoftware.com/articles/detect-app-translocation.html*.

11. See *https://github.com/objective-see/BlockBlock/blob/master/Daemon/Daemon/Plugins/Processes.m*.

12. Patrick Wardle, "Demystifying (& Bypassing) macOS's Background Task Management," presented at DefCon, Las Vegas, August 12, 2023, *https://speakerdeck.com/patrickwardle/demystifying-and-bypassing-macoss-background-task-management*.

13. Phil Stokes, "Backdoor Activator Malware Running Rife Through Torrents of macOS Apps," Sentinel One, February 1, 2024, *https://www.sentinelone.com/blog/backdoor-activator-malware-running-rife-through-torrents-of-macos-apps/*.

14. Jaron Bradley, "Zero-Day TCC Bypass Discovered in XCSSET Malware," Jamf, May 24, 2021, *https://www.jamf.com/blog/zero-day-tcc-bypass-discovered-in-xcsset-malware/*.