Writing a File Monitor with Apple's Endpoint Security Framework

Archived: 2026-04-05 20:10:41 UTC

Writing a File Monitor with Apple's Endpoint Security Framework

by: Patrick Wardle / September 17, 2019


```
# ./fileMonitor
Starting file monitor...[ok]

FILE CREATE ('ES_EVENT_TYPE_NOTIFY_CREATE')

source path: (null)
destination path: /private/tmp/test

process: pid: 849
path: /usr/bin/touch
uid: 501
signing info: {
    cdHash = 818C29925EE42814EFA951413B713788AD62;
    csFlags = 603996161;
    isPlatforBinary = 1;
    signatureIdentifier = "com.apple.touch";
}
```

Background

Earlier this week, I posted a blog titled “Writing a Process Monitor with Apple’s Endpoint Security Framework.” In this post we (rather thoroughly) discussed a new framework/subsystem introduced into macOS Catalina (10.15): “Endpoint Security”

Moreover, we detailed exactly how to build a comprehensive (user-mode) process monitor that leveraged this new framework (and posted the full-source online).

This blog post assumes you've read the previous post, or have a solid understanding of the new Endpoint Security framework.

As such, here, we won't be covering any foundational details about the Endpoint Security framework/subsystem.

A common component of (many) security tools is a file monitor. As its name implies, a file monitor watches for the file I/O events (plus generally extracts information about the process responsible for said file event).

Many of my Objective-See tools contain such a file monitor component and track file events.

Examples include:

- Ransomwhere?
  Tracks file creations to detect the rapid creation of encrypted files by untrusted processes (read: ransomware).

- BlockBlock
  Tracks file creations and modifications in order to detect and alert on persistence events (such as malware installation).

Until now, the preferred way to programmatically create a file monitor in user-mode was to subscribe to events from the `FSEvents` character device (`/dev/fsevents`):

```
1open("/dev/fsevents", O_RDONLY);
```

Though directly reading file events off `/dev/fsevents` , is sufficient (that is to say it provides notifications about file events, and includes the pid of the responsible process) it suffers from various drawbacks and limitations.

First, Apple actually discourages it use (as noted in the `bsd/vfs/vfs_fsevents.c` file):

```
 1if (!strncmp(watcher->proc_name, "fseventsd", sizeof(watcher->proc_name)) ||
 2    !strncmp(watcher->proc_name, "coreservicesd", sizeof(watcher->proc_name)) ||
 3    !strncmp(watcher->proc_name, "mds", sizeof(watcher->proc_name))) {
 4
 5  watcher->flags |= WATCHER_APPLE_SYSTEM_SERVICE;
 6
 7} else {
 8
 9  printf("fsevents: watcher %s (pid: %d) -
10         Using /dev/fsevents directly is unsupported. Migrate to FSEventsFramework\n",
11         watcher->proc_name, watcher->pid);
12}
```

Second, it is rather painful to programmatically interface with, as it it requires one to parse and tokenize various (binary) file events:

```
//skip over args to get to next event struct
-(NSString*)advance2Next:(unsigned char*)ptrBuffer currentOffsetPtr:(int*)ptrCurrentOffset
{
    //path
    NSString* path = nil;

    int arg_len = 0;
    unsigned short *argLen;
    unsigned short *argType;
    struct kfs_event_a *fse;
    struct kfs_event_arg *fse_arg;

    fse = (struct kfs_event_a *)(unsigned char*)
          ((unsigned char*)ptrBuffer + *ptrCurrentOffset);

    //handle dropped events
    if(fse->type == FSE_EVENTS_DROPPED)
    {
        //err msg
        logMsg(LOG_ERR, @"file-system events dropped by kernel");

        //advance to next
        *ptrCurrentOffset += sizeof(kfs_event_a) + sizeof(fse->type);

        //exit early
        return nil;
    }

    *ptrCurrentOffset += sizeof(struct kfs_event_a);
    fse_arg = (struct kfs_event_arg *)&ptrBuffer[*ptrCurrentOffset];

    //save path
    path = [NSString stringWithUTF8String:fse_arg->data];

    //skip over path
    *ptrCurrentOffset += sizeof(kfs_event_arg) + fse_arg->pathlen ;

    argType = (unsigned short *)(unsigned char*)
              ((unsigned char*)ptrBuffer + *ptrCurrentOffset);
    argLen  = (unsigned short *) (ptrBuffer + *ptrCurrentOffset + 2);

    (*argType ==  FSE_ARG_DONE) ? arg_len = 0x2 : arg_len = (4 + *argLen);

    *ptrCurrentOffset += arg_len;

    ...
```

Finally, (and most problematic) though the file events delivered via `/dev/fsevents` contain information about the process responsible for generating the file event (`struct kfs_event_a`), this information is simply a process identifier (`pid`):

```
1typedef struct kfs_event_a {
2    uint16_t type;
3    uint16_t refcount;
4    pid_t    pid;
5} kfs_event_a;
```

When building a comprehensive file monitor (especially as part of a security tool), one generally requires more information about the responsible process, such as its path and code-signing information.

Generating code-signing process via pid, is (somewhat) non-trivial and may also be rather computationally (CPU) intensive.

Although there exist APIs (such as `proc_pidpath`) to generate more comprehensive process information solely from a `pid` , such APIs unsurprisingly fail if the process as (already) terminated. As there is some inherent delay in file events delivered via `/dev/fsevents` , this is actually not uncommon (think malware installers that simply persist a binary then (quickly) exit).

Worse, if the `pid` is reused one may actually mis-identify the process that generated the file event. For a security product, this is rather unacceptable! (For other "issues" with `pids` see: "Don't Trust the PID! ").

It is also possible to receive file I/O events via the OpenBSM subsystem. However, there are limitations to this approach as well, as we highlighted in previous blog post.

As such, until now, the only way to realize a truly effective file monitor was via code running in ring-0 (the kernel).

Apple's Endpoint Security Framework

With Apple's push to kick 3rd-party developers (including security products) out of the kernel, coupled with the realization (finally!) that the existing subsystems were rather archaic and dated, Apple recently announced the new, user-mode "Endpoint Security Framework" (that provides a user-mode interface to a new "Endpoint Security Subsystem").

As we'll see, this framework addresses many of the aforementioned issues & shortcomings!

Specifically it provides a:

- well-defined and (relatively) simple API
- comprehensive process (including code-signing), information for all events
- the ability to proactively respond to file events (though here, our file monitor will be passive).

I'm often somewhat critical of Apple's security posture (or lack thereof). However, the "Endpoint Security Framework" is potentially a game-changer for those of us seeking to write robust user-mode security tools for macOS. Mahalo Apple! Personally I'm stoked 🥳

This blog is practical walk-thru of creating a file monitor which leverages Apple's new framework. For more information on the Endpoint Security Framework, see Apple's developer documentation:

- Endpoint Security Framework

In this blog, we'll illustrate exactly how to create a comprehensive user-mode file monitor that leverages Apple's new framework.

As noted in our previous blog post there are a few prerequisites to leverage the Endpoint Security Framework that include:

- The `com.apple.developer.endpoint-security.client` entitlement
  This can be requested from Apple via this link. Until then (I'm still waiting 😅), give yourself that entitlement (i.e. in your app's `$(ProductName).entitlements` file, and disable SIP such that it remains pseudo-unenforced).

  <dict>
  <key>**com.apple.developer.endpoint-security.client**</key>
  <true/>
  </dict>

- Xcode 11/macOS 10.15 SDK
  As these are both (still) in beta, for now, it's recommended to perform development in a virtual machine (running macOS 10.15, beta).

- macOS 10.15 (Catalina)
  It appears the Endpoint Security Framework will not be made available to older versions of macOS. As such, any tools the leverage this framework will only run on 10.15 or newer.

Ok enough chit-chat, let's dive in!

Our goal is simple: create a comprehensive user-mode file monitor that leverages Apple's new "Endpoint Security Framework".

Besides "capturing" file I/O events, we're also interested in:

- the type of event (create, write, etc.)
- the path(s) of the file ((possibly) source and destination)
- the process responsible for the event, including its:
- process id (pid)

- process path
- any process code-signing information

…luckily, unlike reading events off `/dev/fsevents` the new Endpoint Security framework makes this a breeze!

As noted in our previous blog post, in order to subscribe to events from the "Endpoint Security Subsystem", we must first create a new "Endpoint Security" client. The `es_new_client` function provides the interface to perform this action:

Function

es_new_client

Creates a new client instance and connect it to the Endpoint Security system.

Declaration

```
es_new_client_result_t es_new_client(es_client_t * _Nullable *client, es_handler_
```

Parameters

client
A pointer to receive the new client instance.

handler
The handler to run on all messages sent to this client.

In code, we first include the `EndpointSecurity.h` file, declare a global variable (type: `es_client_t*`), then invoke the `es_new_client` function:

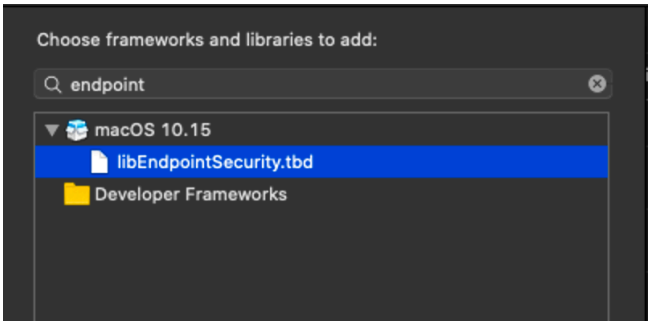
```
#import <EndpointSecurity/EndpointSecurity.h>

//(global) endpoint client
es_client_t* endpointClient = nil;

//create client
// callback invokes (user) callback for new processes
es_new_client(&endpointClient, ^(es_client_t *client, const es_message_t *message)
{
    //process events

});
```

Note that the `es_new_client` function takes an (out) pointer to the variable of type `es_client_t`. Once the function returns, this variable will hold the initialized endpoint security client (required by all other endpoint

security APIs). The second parameter of the `es_new_client` function is a block that will be automatically invoked on endpoint security events (more on this shortly!)

If all is well, the `es_new_client` function will return ES_NEW_CLIENT_RESULT_SUCCESS indicating that it has created a newly initialized Endpoint Security client (`es_client_t`) for us to use.

To compile the above code, link against the Endpoint Security Framework (libEndpointSecurity)



Once we've created an instance of a `es_new_client` , we now must tell the Endpoint Security subsystem what events we are interested in (or want to "subscribe to", in Apple parlance). This is accomplished via the `es_subscribe` function (documented here and in the `ESClient.h` header file):

Function

es_subscribe

Subscribes a client to some set of events.

Declaration

```
es_return_t es_subscribe(es_client_t *client, es_event_type_t *events, uint32_t e
```

Parameters

client
The client to subscribe.

events
An array of event types to subscribe to.

event_count
The number of event types in the array.

This function takes the initialized endpoint client (returned by the `es_new_client` function), an array of events of interest, and the size of said array:

```
 1//(process) events of interest
2es_event_type_t events[] = {
3    ES_EVENT_TYPE_NOTIFY_CREATE,
```

```
    ES_EVENT_TYPE_NOTIFY_WRITE,
    ...
};

//subscribe to events
if(ES_RETURN_SUCCESS != es_subscribe(endpointClient, events,
                            sizeof(events)/sizeof(events[0])))
{
    //err msg
    NSLog(@"ERROR: es_subscribe() failed");

    //bail
    goto bail;
}
```

The events of interest depends on well, what events are of interest to you! As we're writing a file monitor we're (only) interested in file-related events such as:

- `ES_EVENT_TYPE_NOTIFY_CREATE`
  "A type that represents file creation notification events."
- `ES_EVENT_TYPE_NOTIFY_OPEN`
  "A type that represents file opening notification events."
- `ES_EVENT_TYPE_NOTIFY_WRITE`
  "A type that represents file writing notification events."
- `ES_EVENT_TYPE_NOTIFY_CLOSE`
  "A type that represents file closing notification events."
- `ES_EVENT_TYPE_NOTIFY_RENAME`
  "A type that represents file renaming notification events."
- `ES_EVENT_TYPE_NOTIFY_LINK`
  "A type that represents link creation notification events."
- `ES_EVENT_TYPE_NOTIFY_UNLINK`
  "A type that represents link unlinking notification events."

See Apple's [documentation](https://developer.apple.com/documentation/endpointsecurity/es_event_type_t?language=objc) for other events types in the `es_event_type_t` enumeration.

Once the `es_subscribe` function successfully returns (`ES_RETURN_SUCCESS`), the Endpoint Security subsystem will start delivering messages (read: file events).

Recall that the final argument of the `es_new_client` function is a callback block (or handler). Apple states: "The handler block…will be run on all messages sent to this client."

The block is invoked with the endpoint client, and most importantly the message from the Endpoint Security subsystem. This message variable is a pointer of type `es_message_t` (i.e. `es_message_t*`).

Notable members of the `es_message_t` include:

- `es_process_t * process`
  A pointer to a structure that describes the process responsible for the file event.
- `es_event_type_t event_type`
  The type of event (that will match one of the events we subscribed to, e.g. `ES_EVENT_TYPE_NOTIFY_CREATE`)
- `event_type event`
  An event specific structure (i.e. `es_event_create_t`)

Though the `event` member of the `message` structure (`message->event`) is event specific, for all file events it is largely the same. For example, compare the `es_event_write_t` and `es_event_create_t` structures:

```
$ less MacOSX10.15.sdk/usr/include/EndpointSecurity/ESMessage.h

typedef struct {
        es_file_t * _Nullable target;
        uint8_t reserved[64];
} es_event_write_t;

typedef struct {
        es_file_t * _Nullable target;
        uint8_t reserved[64];
} es_event_create_t;
```

Both contain a pointer to a `es_file_t` structure, which contains a path to the file (created, written to, etc.):

```
$ less MacOSX10.15.sdk/usr/include/EndpointSecurity/ESMessage.h

/**
 * es_file_t provides the inode/devno & path to a file that relates to a security event
 * the path may be truncated, which is indicated by the path_truncated flag.
 */
typedef struct {
  es_string_token_t path;
  bool path_truncated;
  union {
          dev_t devno;
          fsid_t fsid;
  };
```

```
  ino64_t inode;
} es_file_t;
```

Note however, some file events (such as the `es_event_rename_t` event) involve both a source and destination file:

```
typedef struct {
  es_file_t * _Nullable source;
  es_destination_type_t destination_type;
  union {
    es_file_t * _Nullable existing_file;
    struct {
            es_file_t * _Nullable dir;
            es_string_token_t filename;
    } new_path;
  } destination;
  uint8_t reserved[64];
} es_event_rename_t;
```

All file events messages also contain a pointer to a `es_process_t` structure (`message->process`) for the responsible process (i.e. that generated the file event). As detailed in our previous post, this structure contains full process information (including pid, path, and code-signing information).

At this point we're stoked as we're receiving all file events along with full details about the responsible process. (No more worrying about pid lookups failing or returning the incorrect process!) 😅

Let's now look illustrate this in code.

First, in the `es_new_client` message callback, we instantiate a (custom) `File` object, passing in the received `es_message_t` message:

```
 1//create client
 2// callback invoked on file events
 3es_new_client(&endpointClient, ^(es_client_t *client, const es_message_t *message)
 4{
 5    //new file obj
 6    File* file = nil;
 7
 8    //init file obj
 9    file = [[File alloc] init:(es_message_t* _Nonnull)message];
10    if(nil != file)
11    {
12        //invoke user callback
13        callback(file);
```

```
14    }
15});
```

This (custom) `File` object's `init:` method simply parses out relevant information from the `es_process_t` structure (such as process id, path, and code-signing information as detailed in our previous post), and then extracts the file path(s):

```
 1//set process
2self.process = [[Process alloc] init:message];
3
4//extract path(s)
5// logic is specific to event
6[self extractPaths:message];
```

The `extractPaths:` method contains event specific logic (as recall some, but not all, file events contain both a source and destination path):

```
 1//extract source & destination path
 2// this requires event specific logic
 3-(void)extractPaths:(es_message_t*)message
 4{
 5  //event specific logic
 6  switch (message->event_type) {
 7
 8    //create
 9    case ES_EVENT_TYPE_NOTIFY_CREATE:
10        self.destinationPath = convertStringToken(&message->event.create.target->path);
11        break;
12
13    //write
14    case ES_EVENT_TYPE_NOTIFY_WRITE:
15        self.destinationPath = convertStringToken(&message->event.write.target->path);
16        break;
17
18    ...
19
20    //rename
21    case ES_EVENT_TYPE_NOTIFY_RENAME:
22
23        //set (src) path
24        self.sourcePath = convertStringToken(&message->event.rename.source->path);
25
26        //existing file ('ES_DESTINATION_TYPE_EXISTING_FILE')
27        if(ES_DESTINATION_TYPE_EXISTING_FILE == message->event.rename.destination_type)
28        {
```

```
29            //set (dest) file
30            self.destinationPath = convertStringToken(&message->event.rename.destination.existing_file->path);
31        }
32        //new path ('ES_DESTINATION_TYPE_NEW_PATH')
33        else
34        {
35            //set (dest) path
36            // combine dest dir + dest file
37            self.destinationPath = [convertStringToken(&message->event.rename.destination.new_path.dir->path)
38        }
39
40        break;
41
42      ...
43  }
44
45  return;
46}
```

Once the `File` object's `init:` method returns, we have a comprehensive (and fully parsed) representation of the reported file event.

File Monitor Library

As noted, several of Objective-See's tools track file events but currently do so via inefficient and (now) antiquated means.

Lucky us, as shown in this blog, we can now leverage Apple's Endpoint Security Subsystem to effectively and comprehensively monitor file events (from user-mode!).

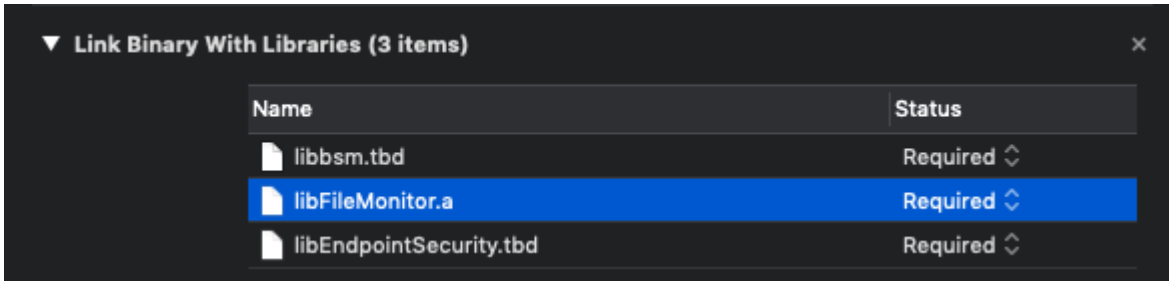
As such, today, I'm releasing an open-source file monitoring library, that implements everything we've discussed here today 🥳

It's fairly simple to leverage this library in your own (non-commercial) tools:

1. Build the library, `libFileMonitor.a`

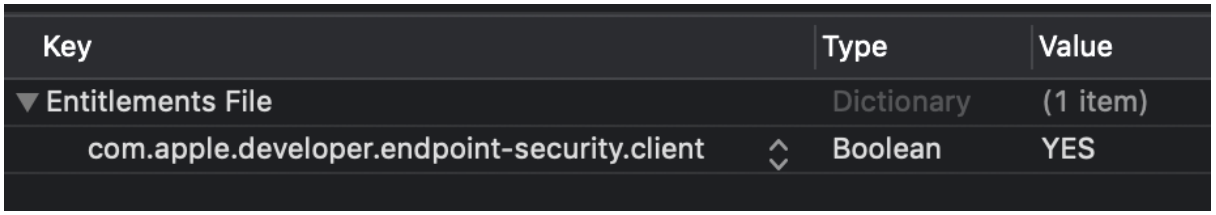
2. Add the library and its header file (`FileMonitor.h`) to your project:

   `#import "FileMonitor.h"`



As shown above, you'll also have link against the `libbsm` (for `audit_token_to_pid`) and `libEndpointSecurity` libraries.

3. Add the `com.apple.developer.endpoint-security.client` entitlement (to your project's `$(ProductName).entitlements` file).



4. Write some code to interface with the library!

This final steps involves instantiating a `FileMonitor` object and invoking the `start` method (passing in a callback block that's invoked on file events). Below is some sample code that implements this logic:

```
//init monitor
FileMonitor* fileMon = [[FileMonitor alloc] init];

//define block
// automatically invoked upon file events
FileCallbackBlock block = ^(File* file)
{
  switch(file.event)
  {
    //create
    case ES_EVENT_TYPE_NOTIFY_CREATE:
        NSLog(@"FILE CREATE ('ES_EVENT_TYPE_NOTIFY_CREATE')");
        break;

    //write
    case ES_EVENT_TYPE_NOTIFY_WRITE:
        NSLog(@"FILE WRITE ('ES_EVENT_TYPE_NOTIFY_WRITE')");
        break;

    //print info
    NSLog(@"%@", file);
};

//start monitoring
// pass in block for events
[fileMon start:block];

//run loop
```

```
29// as don't want to exit
30[[NSRunLoop currentRunLoop] run];
```

Once the `[fileMon start:block];` method has been invoked, the File Monitoring library will automatically invoke the callback (`block`), on file events, returning a `File` object.

The `File` object is declared in the library's header file; `FileMonitor.h` . This object contains information about the file event ((possibly) source and destination path) and the process responsible for the event (in a `Process` object). Take a peek at the `FileMonitor.h` file for more details.

Once compiled, we're ready to start monitoring for file events!

For example, we run: `$ echo "objective-see rules" > /tmp/test` , which generates an open, write, and close file I/O events:

```
# ./fileMonitor
Starting file monitor...[ok]

FILE OPEN ('ES_EVENT_TYPE_NOTIFY_OPEN')
source path: (null)
destination path: /private/tmp/test

process: pid: 649
path: /bin/zsh
uid: 501
signing info: {
    cdHash = BD67298030CA90256B3999A118DCF2FFE5352A9E;
    csFlags = 603996161;
    isPlatforBinary = 1;
    signatureIdentifier = "com.apple.zsh";
}

FILE WRITE ('ES_EVENT_TYPE_NOTIFY_WRITE')
source path: (null)
destination path: /private/tmp/test

process: pid: 649
path: /bin/zsh
uid: 501
signing info: {
    cdHash = BD67298030CA90256B3999A118DCF2FFE5352A9E;
    csFlags = 603996161;
    isPlatforBinary = 1;
    signatureIdentifier = "com.apple.zsh";
}
```

```
FILE CLOSE ('ES_EVENT_TYPE_NOTIFY_CLOSE')
source path: (null)
destination path: /private/tmp/test

process: pid: 649
path: /bin/zsh
uid: 501
signing info: {
    cdHash = BD67298030CA90256B3999A118DCF2FFE5352A9E;
    csFlags = 603996161;
    isPlatforBinary = 1;
    signatureIdentifier = "com.apple.zsh";
}
```

Conclusion

Previously, writing a (user-mode) file monitor for macOS was not a trivial task. Thanks to Apple's new Endpoint Security framework/subsystem (on macOS 10.15+), it's now a breeze!

In short, one simply invokes the `es_new_client` & `es_subscribe` functions to subscribe to events of interest (recalling that the `com.apple.developer.endpoint-security.client` entitlement is required).

For a file monitor, we illustrated how to subscribe to the file-related events such as:

- `ES_EVENT_TYPE_NOTIFY_CREATE`
- `ES_EVENT_TYPE_NOTIFY_OPEN`
- `ES_EVENT_TYPE_NOTIFY_WRITE`

We then showed how to extract the relevant file and (responsible) process structures and parse out all relevant meta-data.

Finally we discussed an open-source file monitoring library that implements everything we've discussed here today. 🥳 \

❤️ Love these blog posts and/or want to support my research and tools? \ You can support them via my [Patreon](https://www.patreon.com/bePatron?c=701171) page! \
