Taking ESF For A(nother) Spin

Cedric Owens

2+ years ago from the date of this blog post I wrote my initial blog post where I started becoming familiar with Apple's Endpoint Security Framework (ESF) on macOS and seeing what different macOS post exploitation activities looked like through the lens of ESF (link: https://medium.com/red-teaming-with-a-blue-team-mentality/taking-the-macos-endpoint-security-framework-for-a-quick-spin-802a462dba06).

Since that time additional offensive techniques have emerged and additional ESF tools have become available so I thought it would be good to revisit this topic. Therefore, this blog will look at some additional post exploitation methods/offensive techniques not covered in my initial blog post.

For all of the test cases below, I will be using Jaron Bradley's ESF Playground tool (https://themittenmac.com/the-esf-playground/). I specifically picked this ESF tool because of how customizable it is and how easy it is to use. To ensure maximum visibility, I followed his steps to enable ALL ESF logs. Then I ran the ESF Playground to have it monitor while performing each test.

Dylib Execution

For this simple test, I used my Dylib Runner tool (https://github.com/cedowens/Dylib_Runner) to execute a “malicious” dylib (that simply launched Calculator.app).

1. First I created runcalc.c to launch Calculator.app:

```
#include <stdio.h>
#include <stdlib.h>int system(const char *command);__attribute
```

2. Next I compiled runcalc.c into a dylib:

```
% gcc -dynamiclib runcalc.c -o RunCalc.dylib
```

3. Then I modified the **RunDylib.swift** source file in my Dylib_Runner repo to execute RunCalc.dylib as follows:

```
import Foundationtypealias addFunc = @convention(c) (CInt,CInt
let myhandle = dlopen(“/Users/dev/RunCalc.dylib”, RTLD_GLOBAL)
```

4. Next I built RunDylib.swift into a macho binary:

```
swiftc -o RunDylib RunDylib.swift
```

5. Lastly, I executed ESF Playground (after giving Terminal full disk access and enabling ALL log sources) and then ran the RunDylib macho (which then executed RunCalc.dylib). The ESF logs that captured this activity are below:

{“event”:”ES_EVENT_TYPE_NOTIFY_LOOKUP”,”file”:{“destination”:"

In particular the **ES_EVENT_TYPE_NOTIFY_MMAP** event type is of interest for detecting dylib load events. The definition of this event type per Apple’s documentation is: **“An identifier for a process that notifies endpoint security that it is mapping a file into memory”** (https://developer.apple.com/documentation/endpointsecurity/es_event_type_t/es_event_type_notify_mmap).

Summary: ESF is seeing the dylib being mapped to memory, which is great. It is worth investigating how noisy *ES_EVENT_TYPE_NOTIFY_MMAP* is in an enterprise environment. My guess is this event type would be very noisy and would need to be tuned by searching for additional indicators.

Dylib Injection (via DYLD_INSERT_LIBRARIES)

Next, we will take a look at what ESF looks like when we inject a dylib. We will do this by searching for an app that either doesn’t have Hardened Runtime enabled or has problematic entitlements (ex: **com.apple.security.cs.disable-library-validation entitlement**). For our example, we will use the Firefox browser.

So, I ran this test using the **DYLD_INSERT_LIBRARIES** environment variable as follows to insert my dylib into Firefox:

```
% DYLD_INSERT_LIBRARIES=~/RunCalc.dylib /Applications/Firefox.
```

Results:

ESF logs also saw the dylib injection and had very similar logs to the test above (Dylib Execution):

{“event”:”ES_EVENT_TYPE_NOTIFY_LOOKUP”,”file”:{“destination”:"

Summary: Once again, the ES_EVENT_TYPE_NOTIFY_MMAP event type is of interest for detecting dylib injection events. This could be a very useful log type with gaining visibility into dylib load/injection events if this can be done across an enterprise in a way that is not too voluminous.

Keylogging

To test keylogging visibility in ESF logs, I used ’s SwiftSpy tool (https://github.com/slyd0g/SwiftSpy) to perform keylogging via IOHIDManager.

Log Results:

{“event”:”ES_EVENT_TYPE_NOTIFY_LOOKUP”,”file”:{**“destination”:**"

The log snippet above is just a representation of the full log samples. After I ran SwiftSpy and allowed it to capture my keystrokes and return to stdout the events above repeated several times. In a nutshell, ESF captures tons of “**ES_EVENT_TYPE_NOTIFY_LOOKUP**”, “**ES_EVENT_TYPE_NOTIFY_OPEN**”, “**ES_EVENT_TYPE_NOTIFY_READDIR**”, and “**ES_EVENT_TYPE_NOTIFY_CLOSE**” events involving **/System/Library/Extensions/IOHIDFamily.kext.**

Summary: The logs above were not generated when I ran ESF Playground and did normal activities on the system so it is worth investigating the prevalence of these ESF events in an enterprise to determine what the noise level is and how to raise the fidelity of building alerting on these log events.

JXA In Memory Execution

This was a test I was really looking forward to running, since this is a popular technique in red team ops for running JXA (JavaScript for Automation) scripts. Fun Times!

For this test I used a Mythic poseidon payload (macho binary built in Go) to run my SwiftBelt-JXA.js JXA post exploitation code in memory. I started ESF monitoring just right before performing the in memory JXA execution using Mythic's jsimport_call function.

Here are the ESF logs from this execution:

{“event”:”ES_EVENT_TYPE_NOTIFY_READLINK”,”file”:{“destination”

Very interesting. The ESF logs showed events related to **JavaScriptAppleEvents.framework**, which fits the bill, as the in memory JXA execution is indeed running JavaScript. You can also see reference to **Foundation.dylib, CoreFoundation.dylib, and Cocoa.dylib** which are imported and used to access certain APIs/functions. It is also interesting that the path to **LaunchServices.dylib** was searched (ES_EVENT_TYPE_NOTIFY_LOOKUP).

Summary: While it may still take some additional analytics to create high fidelity detections for in memory JXA executions, this test did at least identify some visible activities that take place. I wonder if a rule around a source macho binary (or even for osascript) that performs ES_EVENT_TYPE_NOTIFY_READLINK events on JavaScriptAppleEvents.framework and also ES_EVENT_TYPE_NOTIFY_OPEN events on Foundation.dylib, CoreFoundation.dylib, and/or AppKit.dylib in a short period of time may be a decent place to start.

Full Disk Access Check (via sqlite3 open attempt)

Since macOS Privacy Controls (aka, Transparency, Consent, and Control, or TCC) are a core component of macOS security and can pose challenges for attackers and red team operators alike, a while back I put together some code to check for Full Disk Access without generating a TCC prompt. This method simply entailed attempting to open the user's TCC.db and obtain it's size attribute. If the TCC.db size was successfully returned, then it meant my running program had full disk access, and if not then my program did not have full disk access. I added this code to the following repo: https://github.com/cedowens/Spotlight-Enum-Kit/tree/main/TCC-Checker-Swift.

So I compiled the binary above and ran ESF Playground to see what log entries look like for this tactic.

Results:

{"event":"ES_EVENT_TYPE_NOTIFY_LOOKUP","file":{"destination":"

Summary: This activity is captured and seen by ESF event types "ES_EVENT_TYPE_NOTIFY_OPEN" and "ES_EVENT_TYPE_NOTIFY_STAT", both against the user's TCC.db file. It may be worth checking across an enterprise to see how many of these types of events are present and what "proc_path" values are showing for these events (and maybe even looking for less common proc_path values). Since performing file handles are often noisy, I also created another version of this tool that uses the Spotlight database instead to check for FDA access (https://github.com/cedowens/Spotlight-Enum-Kit/blob/main/TCC-Checker-Swift-mdquery/Sources/TCC-Checker-Swift/main.swift)

Spotlight Database Enumeration

This next test will leverage a repo from some research I did a while back on how the Spotlight database can be leveraged for system enumeration and to find sensitive files (blog post: https://cedowens.medium.com/spotlighting-your-tcc-access-permissions-ec6628d7a876). In a nutshell, I wrote a set of tools that make spotlight database API calls (i.e., MDQuery) to check TCC permissions to ~/Desktop, ~/Documents, and ~/Downloads, check for full disk access, search for recently created/modified files, and search for files of interest with keywords on the system. All of these tools exclusively search the spotlight database for this information, so let's see what ESF logs look like.

1. I compiled and executed the TCC-Checker-Swift project (at https://github.com/cedowens/Spotlight-Enum-Kit/blob/main/TCC-Checker-Swift-mdquery/Sources/TCC-Checker-Swift/main.swift).
2. I ran ESF Playground and filtered for logs related to this binary.

Results:

Unfortunately ESF logs only captured "**ES_EVENT_TYPE_NOTIFY_FSGETPATH**" event types related to the Spotlight database, which probably do not provide strong enough data to build detections around when it comes to Spotlight database queries/enumeration. **ES_EVENT_TYPE_NOTIFY_FSGETPATH means that the source file is retrieving a file system path (in the "destination" field).** Example ESF log capture:

```
{"event":"ES_EVENT_TYPE_NOTIFY_FSGETPATH","file":{"destinatior
```

Summary: The logs show several ES_EVENT_TYPE_NOTIFY_FSGETPATH events for ~/Library/Containers/com.apple.CoreSpotlight.CoreSpotlightImportExtension. However, the spotlight db queries themselves were not

present in the ESF logs, which is interesting and seems to be a visibility limitation. However, if I missed something here and if anyone does find a way to view this data in ESF or from other sources please do reach out!

Overall I was surprised with ESF's visibility into some of the tactics/techniques above. I think ESF is solid and offers some good potential for building detections around. However, there seems to be little information publicly around correlating different event types and creating high fidelity macOS detections using ESF for enterprise macOS deployments so I hope that this post can at least help generate more discussions on that topic. Thanks for reading!