Beyond the good ol' LaunchAgents - 23 - emond, The Event Monitor Daemon

theevilbit.github.io/beyond/beyond_0023

November 27, 2021

> This is part 23 in the series of "Beyond the good ol' LaunchAgents", where I try to collect various persistence techniques for macOS. For more background check the introduction.

This post will be about `emond`, Apple's Event Monitor daemon.

I think almost everything has been already told about this method and `emond` in general by James Reynolds here and xorrior here so really not much left for me. There is no point for me replicating their awesome posts, so please just read them. That's it! Thank you for reading! ¯\(ツ)/¯ .

.

.

.

.

.

.

.

.

OK, well... I will still give a super-brief summary of how to persist with emond, and then, I decided to add my contribution to the `emond` documentation, so stay tuned.

The TL;DR

The `emond` service is defined in `/System/Library/LaunchDaemons/com.apple.emond.plist`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Disabled</key>
        <false/>
        <key>Label</key>
        <string>com.apple.emond</string>
        <key>MachServices</key>
        <dict>
                <key>com.apple.emond.evtq</key>
                <dict>
                        <key>HideUntilCheckIn</key>
                        <true/>
                </dict>
        </dict>
        <key>Program</key>
        <string>/sbin/emond</string>
        <key>ProgramArguments</key>
        <array>
                <string>/sbin/emond</string>
        </array>
        <key>QueueDirectories</key>
        <array>
                <string>/private/var/db/emondClients</string>
        </array>
        <key>DrainMessagesOnFailedInit</key>
        <true/>
        <key>EnablePressuredExit</key>
        <true/>
        <key>EnableTransactions</key>
        <false/>
</dict>
</plist>
```

It will start, whenever there is a file in the directory `/private/var/db/emondClients/`, which is specified by `QueueDirectories` in the PLIST file. We can create even an empty file, the content is not so important.

```
sudo touch /private/var/db/emondClients/some
```

Then `emond` will started by `launchd` and it will process the rules found inside `/etc/emond.d/rules/`. These rules define what action to perform when a specific event occurs. For a detailed explanation of these events and actions please refer to the blog posts I linked at the very beginning. For the persistence example I will show how we can run a simple command when `emond` is starting up. First, let's make a copy of the default sample rule.

```
sudo cp /etc/emond.d/rules/SampleRules.plist /etc/emond.d/rules/startup.plist
```

and edit it so it looks like this.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
        <dict>
                <key>name</key>
                <string>create file</string>
                <key>enabled</key>
                <true/>
                <key>eventTypes</key>
                <array>
                        <string>startup</string>
                </array>
                <key>actions</key>
                <array>
                        <dict>
                                <key>command</key>
                                <string>/bin/bash</string>
                                <key>user</key>
                                <string>root</string>
                                <key>arguments</key>
                                <array>
                                        <string>-c</string>
                                        <string>touch
/Library/emondstarted.txt</string>
                                </array>
                                <key>type</key>
                                <string>RunCommand</string>
                        </dict>
                </array>
        </dict>
</array>
</plist>
```

Here we define a command that will be executed when `emond` starts (`eventTypes`:`startup`). Note that `emond` will startup during boot time, when the user is not yet logged in. We can control the command with the `command`, `arguments` and `user` keys. The above will execute `/bin/bash -c touch /Library/emondstarted.txt` as root. Note that the `enabled` key is set to `true`.

Now if we restart `emond` by killing it or rebooting our machine the above command will be executed and the file `/Library/emondstarted.txt` will be created.

The `/etc/emond.d/emond.plist` file contains configuration options, like other rule directories (`additionalRulesPaths`) and also the following filter:

```
<key>filterByUID</key>
<string>0</string>
<key>filterByGID</key>
<string></string>
```

This filter controls who can call emond's XPC service, what we will discuss next.

com.apple.emond.evtq

I decided to dig into a bit into the XPC internals of `emond`, which I think was not documented before.

As defined in the launchd file, the XPC service name is `com.apple.emond.evtq`. If we open `emond` in a disassembler, we can find that it's created here:

```
/* @class Monitor */
-(int)SetXPCListenerFor:(void *)arg2 {
    r15 = arg2;
    r14 = self;
    rax = xpc_connection_create_listener("com.apple.emond.evtq", arg2, arg2);
    if (rax != 0x0) {
            xpc_connection_set_legacy(rax);
            var_48 = *__NSConcreteStackBlock;
            *(&var_48 + 0x8) = 0xffffffffc2000000;
            *(&var_48 + 0x10) = sub_10000e621;
            *(&var_48 + 0x18) = 0x10002c420;
            *(&var_48 + 0x20) = r14;
            *(&var_48 + 0x28) = r15;
            xpc_connection_set_event_handler(rax, &var_48);
            *(r14 + 0x18) = rax;
            xpc_retain(rax);
            xpc_connection_resume(rax);
            rax = 0x0;
    }
    else {
            rax = 0xffffffffffffffff;
    }
    return rax;
}
```

Inside the connection handler we find a call to `filterBySourceUID:GID:`.

```
int sub_10000e621(int arg0, int arg1, int arg2, int arg3, int arg4, int arg5) {
...
            r15 = xpc_connection_get_euid(r13);
            rax = xpc_connection_get_egid(r13);
            r12 = rax;
            sub_10000bbc5(0x3, "XPC Connection Request recieved from %d:%d
(uid:gid)", r15, rax, r8, r9, var_60);
            if ([*(r14 + 0x20) filterBySourceUID:r15 GID:r12] != 0x0) {
...
```

The actual UID/GID filter is populated at the `EntryPoint`.

```
*qword_100038cb8 = [[Monitor alloc] initWithRulebase:*qword_100038ca8
andVariables:r13];
[*qword_100038cb8 setEventFilterUID:sub_10000c060([*qword_100038ca0
objectForKey:@"filterByUID", r13]) FilterGID:sub_10000c060([*qword_100038ca0
objectForKey:@"filterByGID", r13])];
```

Without going into further details, if we follow these function, we will find that it parses the config file, and reads in the values defined in `filterByUID` and `filterByGID`. This means that by default only `root` level processes can communicate with `emond`, but we can also modify this setting if needed, allowing other processes.

So we know that only root can connect. But how normally messages are sent? The `/System/Library/LaunchDaemons/com.apple.emlog.plist` defines a perl script at `/usr/libexec/emlog.pl`. This script uses the `/usr/libexec/xssendevent` binary to send messages to `emond`.

For example:

```
echo "{ eventType = auth.failure; eventSource = emlog.pl; eventDetails = {clientIP =
\"1.1.1.1\"; hostPort = 21; protocolName = \"ftp\";};}" | /usr/libexec/xssendevent
```

Since only root can send messages, we need to execute this command as root.

I edited my `/private/etc/emond.d/rules/SampleRules.plist` to log a message for the `auth.failure` event type.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
        <dict>
                <key>name</key>
                <string>sample rule</string>
                <key>enabled</key>
                <true/>
                <key>eventTypes</key>
                <array>
                        <string>auth.failure</string>
                </array>
                <key>actions</key>
                <array>
                        <dict>
                                <key>message</key>
                                <string>Event Monitor started at ${builtin:now}
</string>
                                <key>type</key>
                                <string>Log</string>
                                <key>logLevel</key>
                                <string>Notice</string>
                                <key>logType</key>
                                <string>syslog</string>
                        </dict>
                </array>
        </dict>
</array>
</plist>
```

Now if we execute the command above, we will see a log message in the log stream.

```
sh-3.2# echo "{ eventType = auth.failure; eventSource = emlog.pl; eventDetails =
{clientIP = \"1.1.1.1\"; hostPort = 21; protocolName = \"ftp\";};}" |
/usr/libexec/xssendevent

-----------

csaby@mac ~ % log stream | grep "Event Monitor"
2021-11-27 14:47:02.926999+0100 0x66a7f3   Default     0x0                  18157  0
emond: Event Monitor started at 659713622.926977
```

We can save the event in a json plist file, and specify it as the input for `xssendevent`. The json formatted event PLIST file:

```
{ eventType = auth.failure; eventSource = emlog.pl; eventDetails = {clientIP =
"1.1.1.1"; hostPort = 21; protocolName = "ftp";};}
```

Then using the `-e` option, we can specify the previously saved file.

```
/usr/libexec/xssendevent -e event.plist
```

But how can we do this directly with xpc? `xssendevent` uses the `/usr/lib/libXSEvent.dylib` library to send a message.

```
sh-3.2# otool -L /usr/libexec/xssendevent
/usr/libexec/xssendevent:
        /usr/lib/libXSEvent.dylib (compatibility version 1.0.0, current version
1.0.0)
        /System/Library/Frameworks/Security.framework/Versions/A/Security
(compatibility version 1.0.0, current version 60157.40.30)
        /System/Library/Frameworks/Foundation.framework/Versions/C/Foundation
(compatibility version 300.0.0, current version 1855.103.0)
        /usr/lib/libobjc.A.dylib (compatibility version 1.0.0, current version
228.0.0)
        /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version
1311.0.0)
        /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
(compatibility version 150.0.0, current version 1855.103.0)
```

Instead of reversing everything, we can use Jonathan Levin's `XPoCe` utility to sniff the XPC communication, and then recreate the code ourselves. This is much easier as we will see the messages directly without doing lot's of manual reversing.

To do this we need a machine with SIP disabled, as `XPoCe` injects code into the target process. Then we can send our event again.

```
sh-3.2#  echo "{ eventType = auth.failure; eventSource = emlog.pl; eventDetails =
{clientIP = \"1.1.1.1\"; hostPort = 21; protocolName = \"ftp\";};}" | ./XPoCe
/usr/libexec/xssendevent
xpc_connection_create ( "com.apple.emond.evtq",0x7f9c3640a0f0)  = 0x7f9c36504080 ()
xpc_connection_send_message ( connection@0x7f9c36504080,dictionary@0x7f9c3640b070)

 = "<connection: 0x7f9c36504080> { name = com.apple.emond.evtq, listener = false, pid
= 0, euid = 4294967295, egid = 4294967295, asid = 4294967295 }"
 = "<dictionary: 0x7f9c3640b070> { count = 3, transaction: 0, voucher = 0x0, contents
=
        "eventDetails" => <dictionary: 0x7f9c3640b0d0> { count = 4, transaction: 0,
voucher = 0x0, contents =
                "clientIP" => <string: 0x7f9c3640b1f0> { length = 7, contents =
"1.1.1.1" }
                "protocolName" => <string: 0x7f9c3640b290> { length = 3, contents =
"ftp" }
                "hostPort" => <string: 0x7f9c3640b260> { length = 2, contents = "21"
}
                "eventSource" => <string: 0x7f9c3640b370> { length = 8, contents =
"emlog.pl" }
        }
        "eventType" => <string: 0x7f9c3640b340> { length = 12, contents =
"auth.failure" }
        "eventTimestamp" => <double: 0x7f9c3640b130>: 659713961.621497
}"
```

We can easily translate the above output to the following C code suing the standard XPC libraries.

```
#include <stdio.h>
#include <stdlib.h>
#include <xpc/xpc.h>

int main(int argc, const char **argv) {

    xpc_connection_t service;
    xpc_object_t msg, event_details;

    msg = xpc_dictionary_create(NULL, NULL, 0);
    event_details = xpc_dictionary_create(NULL, NULL, 0);

    xpc_dictionary_set_string(event_details, "clientIP", "1.1.1.1");
    xpc_dictionary_set_string(event_details, "protocolName", "ftp");
    xpc_dictionary_set_string(event_details, "hostPort", "21");
    xpc_dictionary_set_string(event_details, "eventSource", "myxpc");

    xpc_dictionary_set_string(msg, "eventType", "auth.failure");
    xpc_dictionary_set_value(msg, "eventDetails", event_details);
    xpc_dictionary_set_double(msg, "eventTimestamp", 659713961.621497);


    service = xpc_connection_create_mach_service("com.apple.emond.evtq", NULL, 0);
    if (service == NULL) {
        perror("xpc_connection_create_mach_service");
    }

    xpc_connection_set_event_handler(service, ^(xpc_object_t event) {
        xpc_type_t type = xpc_get_type(event);
        if (type == XPC_TYPE_ERROR)
        {
            printf("emond, client, xpc error: %s", xpc_dictionary_get_string(event,
XPC_ERROR_KEY_DESCRIPTION));
        }
        else
        {
            char* description = xpc_copy_description(event);
            printf("emond, client, xpc unexpected type: %s", description);

        }
    });


    xpc_connection_resume(service);

    xpc_connection_send_message(service, msg);

}
```

Then, once we compile it and execute, we can verify that our event message was properly logged by `emond`, by our rule.

```
csaby@mac emond % sudo ./emondxpc

------------------------------

csaby@mac ~ % log stream | grep "Event Monitor"
2021-11-27 15:09:50.539508+0100 0x66de62   Default     0x0                  18157  0
emond: Event Monitor started at 659714990.539463
```

If we set the “eventType” to “startup” we can get our file created.

What it is good for? I don’t know :))) I don’t think it’s good for any privilege escalation as we can’t communicate it with `emond` as a regular user and `emond` doesn’t have any cool entitlements. The effectiveness of our event message is also tied to the rules configured on the machine, which is nothing by default. However we might find an environment where `emontd` is configured differently, and maybe there is some interesting rule that we can trigger. Other than that I just entertained myself with some reverse engineering and get to know `emond` a bit better.