VB2021
localhost

7 - 8 October, 2021 / vblocalhost.com

MITIGATING EXPLOITS USING APPLE'S ENDPOINT SECURITY

Csaba Fitzl

Offensive Security, Hungary

fitzl.csaba@gmail.com

ABSTRACT

I have spent the last two years searching for logic vulnerabilities in both *Apple*'s *macOS* operating system and third-party apps running on *macOS*. One of the common ways to gain more privileges is by injecting code into another process that possesses various entitlements, which grants various rights to the process. Although *Apple*'s own processes are well protected the same is not true for third-party apps. This has opened up the possibilities for plenty of privacy (TCC) related bypasses and privilege escalation to root through XPC services.

Another common logic vulnerability is to attack the system and applications through symbolic or hard links by redirecting the file operation to a location of the attacker's choice.

When *Apple* introduced the Endpoint Security framework, I decided to write an application to protect against such logic attacks, and to learn the framework myself. The application, called Shield [1], is free and open source [2].

In this paper I will introduce the basic concepts behind process injection and file link attacks. I will talk about how they work, and what they make possible. Then I will discuss *Apple*'s Endpoint Security framework, how it works, and how it can be used.

Next, I will discuss the development of the Shield application, how the mitigations are implemented, and how it works in the background. I will also describe my experiences of getting the Endpoint Security entitlement from *Apple*.

PROCESS INJECTION ATTACKS ON MACOS

Process injection [3] is a common attack technique on all operating systems. On *Windows* it's mainly used for stealth and to hide code in a different process, and it's also generally allowed between processes running on the same privilege (ring) level. This means that a user-mode process can typically inject code into another user-mode process.

On *macOS* the situation is entirely different, and there is a reason for that. On *macOS*, processes' code signatures control access to various resources through entitlements. Entitlements are strings that typically take the form of reverse DNS notation, and are stored inside the code signature of the application. All privacy-related access is tied to various entitlements. *Apple* itself has over 200 private entitlements that granularly control what a process can access in the system, and these give *Apple* binaries unique rights. For example, the `kernelmanagerd` daemon, which is used to load and unload kernel extensions, has the following entitlements:

```
Executable=/usr/libexec/kernelmanagerd
...
<dict>
        <key>com.apple.private.KextAudit.user-access</key>
        <true/>
        <key>com.apple.private.allow-bless</key>
        <true/>
        <key>com.apple.private.applecredentialmanager.allow</key>
        <true/>
        <key>com.apple.private.iokit.nvram-panicmedic</key>
        <true/>
        <key>com.apple.private.iokit.system-nvram-allow</key>
        <true/>
        <key>com.apple.private.kernel.get-kext-info</key>
        <true/>
        <key>com.apple.private.security.bootpolicy</key>
        <true/>
        <key>com.apple.private.security.iocatalog-management</key>
        <true/>
        <key>com.apple.private.security.kext-collection-management</key>
        <true/>
        <key>com.apple.private.security.kext-management</key>
        <true/>
        <key>com.apple.private.security.storage.SystemExtensionManagement</key>
        <true/>
        <key>com.apple.private.security.syspolicy.kext-management</key>
        <true/>
        <key>com.apple.private.spawn-driver</key>
        <true/>
        <key>com.apple.private.storagekitd.statuschange</key>
```

```
        <true/>
        <key>com.apple.private.system-extensions.extension-point</key>
        <true/>
        <key>com.apple.private.tcc.allow</key>
        <array>
                <string>kTCCServiceSystemPolicyAllFiles</string>
        </array>
        <key>com.apple.private.xpc.launchd.job-manager</key>
        <string>com.apple.kernelmanagerd</string>
        <key>com.apple.rootless.storage.KernelExtensionManagement</key>
        <true/>
        <key>com.apple.rootless.storage.KernelExtensionStaging</key>
        <true/>
        <key>com.apple.rootless.volume.Preboot</key>
        <true/>
        <key>com.apple.rootless.volume.iSCPreboot</key>
        <true/>
</dict>
</plist>
```

The entitlement `com.apple.private.security.kext-management` gives the `kernelmanagerd` process the ability to load kernel extensions. The concept of code signatures and entitlements means that if we can run code in the context of another process, we get access to its entitlements and we will gain the same rights. Beyond entitlements, keychain access and secure XPC-based interprocess communications are tied to the processes' code signatures. For example, in the case of XPC we can invoke the privileged XPC service on behalf of the client, which often results in local privilege escalation.

Again, if we can impersonate the application we will gain extended privileges. Because of this, process injection is locked down on *macOS*.

Apple binaries are protected by System Integrity Protection (SIP) and disallow all forms of code injection. Applications compiled with hardened runtime also enjoy the protection of SIP. However, there are still plenty of third-party applications that either don't have hardened runtime enabled, or use entitlements which effectively disable code injection protection.

Next we will review the most common methods that can be used to inject code into another process on *macOS*.

Using the DYLD_INSERT_LIBRARIES environment variable

This technique is probably the most classic way of injecting code into another process. One can use the `DYLD_INSERT_LIBRARIES` environment variable, specify a dylib in it, and that dylib will be loaded into the application before it even starts. I wrote about this in detail in [4] and [5].

For example, the following line will inject `inject.dylib` into the application `test`:

```
% DYLD_INSERT_LIBRARIES=inject.dylib ./test
```

This technique had some limitations even in the initial versions of *OS X*. For example, the injection wasn't permitted on binaries that had the SUID bit set or those with a `__RESTRICTED` segment on the Mach-O file. Nowadays, SIP-protected binaries are all protected from this injection attack. However, there is a way to opt out of this protection: an application using the `com.apple.security.cs.allow-dyld-environment-variables` entitlement will allow this type of injection. If possible, developers should avoid using this entitlement.

Dylib hijacking and proxying

Dylib hijacking was discussed in detail by Patrick Wardle in 2015 [6]. The basic idea of this attack is that we plant a rogue dylib on the system, which will be loaded by the application.

If we hijack the dylib search order we call the attack 'dylib hijacking'. In this case the application first tries to load the dylib from paths where the dylib is not found. If we plant our dylib in these locations our dylib will be loaded. If we swap an existing dylib, we call it dylib proxying. Generally speaking, we will re-export the functions of the original dylib in both cases, and proxy the original function calls, but in the second case we will need to swap an existing file.

This attack is prevented if the main executable is compiled with library validation. Library validation means that an application can only load shared libraries that have been signed with the same team ID or *Apple*. SIP will also enforce library validation. There are cases when developers need to support third-party libraries, such as plug-ins or drivers, and in those cases they can use the `com.apple.security.cs.disable-library-validation` entitlement, which will disable library validation. Having this entitlement will generically allow us to inject code into the app, thus it's extremely dangerous. The use of this entitlement should be avoided.

Injection through task ports

If we want to gain access to and control a given process or thread we need to gain access to its underlying Mach task's 'task port'. A task is a unit defined by the Mach microkernel, which was originally developed by Carnegie Mellon University, and it is used in *macOS* as well. Since task ports allow unrestricted control over a process, once we have retrieved the task port, we can perform any action we would like on the process. This means we can read and write its memory, we can create threads, terminate it, etc. Effectively, we gain full control, which will allow us to inject our code into the target process.

Task port access is also restricted on any process that runs under SIP's protection. Similarly to the other protections, this can be also disabled through an entitlement, which is `com.apple.security.get-task-allow`, and as such this entitlement is extremely dangerous. However, there are cases when it is required. For example, if we want to debug the application we develop, we need to add this entitlement in order to be able to attach a debugger to the process. Because of this, Xcode will automatically add this entitlement for any debug build. Unfortunately, there have been cases in the past where developers distributed their debug build instead of the release build, which resulted in a vulnerable application. These days, *Apple*'s notarization service will reject any application that has this entitlement. Fortunately, as notarization is almost mandatory nowadays, we have fewer and fewer applications with the wrong entitlements.

Electron applications

The last group of injections we need to mention are specifically related to *Electron* applications. *Electron* applications are infamous for their poor security, as has been discussed before by various researchers [7, 8]. The main concern is that *Electron*'s security model, which is based on *Chromium*'s, doesn't deal with local attacks, thus it doesn't try to prevent local process injection. This makes almost all of these apps vulnerable. This is a problem, as there are hundreds of these applications [9], including some very well known ones such as *Discord*, *Slack*, *Microsoft Teams* and *WhatsApp* to just name a few.

Attackers have multiple ways of injecting code into *Electron* apps. One is by setting the `ELECTRON_RUN_AS_NODE` environment variable to `1`, which will cause the application start as a Node.js console, and custom JavaScript can be run in the context of the application. This allows for trivial code injection.

Another method is to start the *Electron* application with the debug option `(--inspect)` set, which enables a debugger to be attached, and thus for custom JavaScript code to be injected. This again makes code injection easy.

Attackers can use either the `codesign` utility or code signing related APIs to enumerate targets. Although the entitlements of *Electron* apps varies they often have access to the user's microphone or camera. For example, *Discord* possesses the following entitlements:

```
<dict>
        <key>com.apple.security.cs.allow-unsigned-executable-memory</key>
        <true/>
        <key>com.apple.security.device.audio-input</key>
        <true/>
        <key>com.apple.security.device.camera</key>
        <true/>
</dict>
```

As this allows an attacker to bypass *Apple*'s privacy protection once injected into the application, it can be an attractive target for local exploitation.

Unfortunately, SIP can't prevent these attacks.

Vulnerabilities

As we can see there are a wide variety of options to inject code into other processes on *macOS*. Luckily, SIP prevents most of these. There are cases, however, where hardened runtime is missing or insecure entitlements are used, which disable SIP's protective umbrella. These can often lead to serious consequences. Here is a list of a few vulnerabilities that were possible due to process injection:

- CVE-2020-26893 – *ClamXAV AntiVirus* XPC Local Privilege Escalation
  - By injecting our code into the *ClamXAV* AV client we could communicate with *ClamXAV*'s privilege helper tool, which allowed us to escalate our privileges to root.
- CVE-2020-29621 – `coreaudiod` TCC bypass
  - By injecting our own plug-in into the `coreaudiod` process we gained access to the process's entitlements, including `com.apple.private.tcc.manager`, which allowed us to fully control the system privacy settings and completely bypass TCC.

- CVE-2020-25736 – *Acronis True Image 2021* XPC Local Privilege Escalation
  - By injecting our code into the *Acronis True Image* AV client we could communicate with *Acronis True Image*'s privilege helper tool, which allowed us to escalate our privileges to root.
- CVE-2020-24259 – *Signal macOS* TCC bypass
  - By performing a dylib proxying attack we were able to gain access to *Signal*'s TCC permissions, which allowed us to access the microphone without any user prompt, effectively partially bypassing TCC.
- CVE-2020-14978 – *F-Secure* XPC business logic compromise
  - By injecting our code into *F-Secure* AV client we could communicate with *F-Secure*'s privilege helper tool, which allowed us to control the otherwise protected AV.

This is just a small subset of the issues. Based on the list above we can see why process injection is generically forbidden on *macOS*, and if it happens it can easily lead to various vulnerabilities ranging from simple privacy bypasses to full privilege escalations.

Next we cover another group of logic attacks, which utilizes file links.

FILE LINK ATTACKS ON MACOS

File link attacks are another big group of logic attacks on *macOS*. I discussed this in depth in [10].

Generally, file link attacks work by someone placing a symbolic or hard link in a specific, user-controllable location, which will then be followed by a process running with higher privileges, typically root. Depending on what the process does, this can lead to various scenarios:

- Arbitrary file overwrite
- Arbitrary file deletion
- Full local privilege escalation

As we only redirect the file operation to a different location, and we don't have direct control over the target file, we usually need to find a way to indirectly control the target file. This is not always possible.

Vulnerabilities

Because privileged processes often work on unprivileged locations, we can exploit these scenarios. The following is a list of a few vulnerabilities, which were all related to file link attacks:

- CVE-2020-9900 – Crash Reporter Local Privilege Escalation
  - This vulnerability allowed an attacker to copy a script into the periodic scripts folder, which was later executed as root.
- CVE-2020-3855 – *macOS* DiagnosticMessages arbitrary file overwrite
  - Because DiagnosticMessages was writing to log files under user-controllable locations as root, it allowed someone to overwrite arbitrary files.
- CVE-2020-3762 – *Adobe* installer Local Privilege Escalation
  - Because the *Adobe* installer was incorrectly handling files in the `/tmp/` directory, it allowed an attacker to escalate privileges to root by modifying files, which were later moved to the `Library/LaunchDaemons` directory.
- CVE-2021-1786 – Crash Reporter arbitrary file deletion
  - This vulnerability allowed someone to delete arbitrary files as root.

As we can see, placing simple file links can easily lead to serious issues.

Next we will discuss the Mandatory Access Control Framework, which makes the Endpoint Security framework really powerful.

MANDATORY ACCESS CONTROL FRAMEWORK – MACF

The Mandatory Access Control Framework has its origin in TrustedBSD and has been adopted by *Apple*. It was introduced in *Mac OS X 10.5* (*Leopard*). It's a policy framework which provides the ability to extend the kernel with various policy-based modules. Its high level architecture is shown in Figure 1.

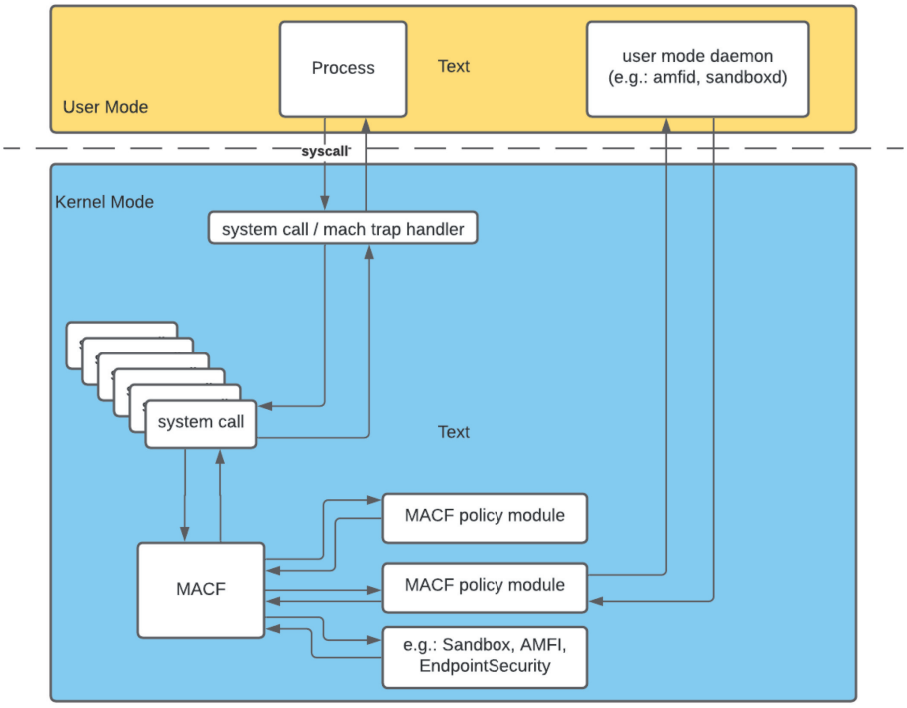

Figure 1: High-level architecture of the Mandatory Access Control Framework.

The kernel implements MAC policy hooks in various system calls or Mach traps (Mach system calls), which at the time of this writing means over 200 possible hooks. Policy modules come in the form of a kernel extension (KEXT), and when they are installed, they register their name and the list of supported hooks.

When a system call happens, the kernel iterates through all the MAC policy modules, and if one implements the hook, the kernel will make a callout. Policy modules can make a decision as to whether an action is allowed or not for the given caller from user space.

Due to the large number of possible hooks, the framework is extremely powerful, and *Apple* itself uses it in security-related kernel extensions, like AppleMobileFileIntegrity, Sandbox, EndpointSecurity and Quarantine. These modules hook various numbers of functions, with Sandbox containing the most interception points.

Unfortunately, third-party MAC drivers were never officially supported by *Apple*. Although the `mac.h` header was part of the kernel development kit (KDK), it was removed as of *macOS 10.12*. Although the header is still available through the XNU source codes, and developers can implement MAC policy modules, it is not officially supported.

Let's take a brief look at how MAC policy hooks are implemented. Below we have a snippet from the `snapshot_mount` system call:

```
#if CONFIG_MACF
    error = mac_mount_check_snapshot_mount(ctx, rvp, vp, &dirndp->ni_cnd, snapndp->ni_cnd.cn_
nameptr,
        mp->mnt_vfsstat.f_fstypename);
    if (error) {
        goto out2;
    }
#endif
```

The system call has a callout to the MAC framework through the function `mac_mount_check_snapshot_mount`. The MAC callouts are typically named as `mac_*`. Let's follow this function, its source code is shown below:

```
mac_mount_check_snapshot_mount(vfs_context_t ctx, struct vnode *rvp, struct vnode *vp, struct
componentname *cnp,
        const char *name, const char *vfc_name)
{
        kauth_cred_t cred;
        int error;

#if SECURITY_MAC_CHECK_ENFORCE
        /* 21167099 - only check if we allow write */
```

```
        if (!mac_vnode_enforce) {
                return 0;
        }
#endif
        cred = vfs_context_ucred(ctx);
        if (!mac_cred_check_enforce(cred)) {
                return 0;
        }
        VFS_KERNEL_DEBUG_START1(92, vp);
        MAC_CHECK(mount_check_snapshot_mount, cred, rvp, vp, cnp, name, vfc_name);
        VFS_KERNEL_DEBUG_END1(92, vp);
        return error;
}
```

`mac_mount_check_snapshot_mount` will perform a couple of checks and make a call to `MAC_CHECK`, which is a C macro. This is where the actual MAC policy modules will be called.

```
#define MAC_CHECK(check, args...) do {                                  \
        struct mac_policy_conf *mpc;                                    \
        u_int i;                                                        \
                                                                        \
        error = 0;                                                      \
        for (i = 0; i < mac_policy_list.staticmax; i++) {               \
                mpc = mac_policy_list.entries[i].mpc;                   \
                if (mpc == NULL)                                        \
                        continue;                                       \
                                                                        \
                if (mpc->mpc_ops->mpo_ ## check != NULL)                \
                        error = mac_error_select(                       \
                            mpc->mpc_ops->mpo_ ## check (args),         \
                            error);                                     \
        }                                                               \
        if (mac_policy_list_conditional_busy() != 0) {                  \
                for (; i <= mac_policy_list.maxindex; i++) {            \
                        mpc = mac_policy_list.entries[i].mpc;           \
                        if (mpc == NULL)                                \
                                continue;                               \
                                                                        \
                        if (mpc->mpc_ops->mpo_ ## check != NULL)        \
                                error = mac_error_select(               \
                                    mpc->mpc_ops->mpo_ ## check (args), \
                                    error);                             \
                }                                                       \
                mac_policy_list_unbusy();                               \
        }                                                               \
} while (0)
```

The `MAC_CHECK` macro will iterate over all MACF policy extensions, and call the policy module's function if it implements it. There is a callout to the `mac_error_select` function, which will select the overall return value based on all the responses from all modules. If any module denies the action, the end result will be denied.

The actual policy hooks are named as `mpo_*`, and we can find below the ones responsible for snapshot mounting:

```
typedef int mpo_mount_check_snapshot_mount_t(
        kauth_cred_t cred,
        struct vnode *rvp,
        struct vnode *vp,
        struct componentname *cnp,
        const char *name,
        const char *vfc_name
        );
```

Overall, MAC is an extremely powerful framework to control security on a very granular basis. Next we will discuss the Endpoint Security framework.

THE ENDPOINT SECURITY FRAMEWORK

Apple introduced the Endpoint Security framework [11] in *macOS Catalina* (10.15) to provide security software with a user-mode API to protect against cyber attacks. At the same time, *Apple* started to deprecate various kernel APIs to force vendors out of the kernel space and to use the new API instead.

The Endpoint Security framework has been discussed in detail by Scott Knight in [12] and at the Objective by the Sea conference [13]. Here I will just provide a quick summary, for further details please refer to Scott's work.

The architecture of the framework can be illustrated as follows (made by Scott Knight).

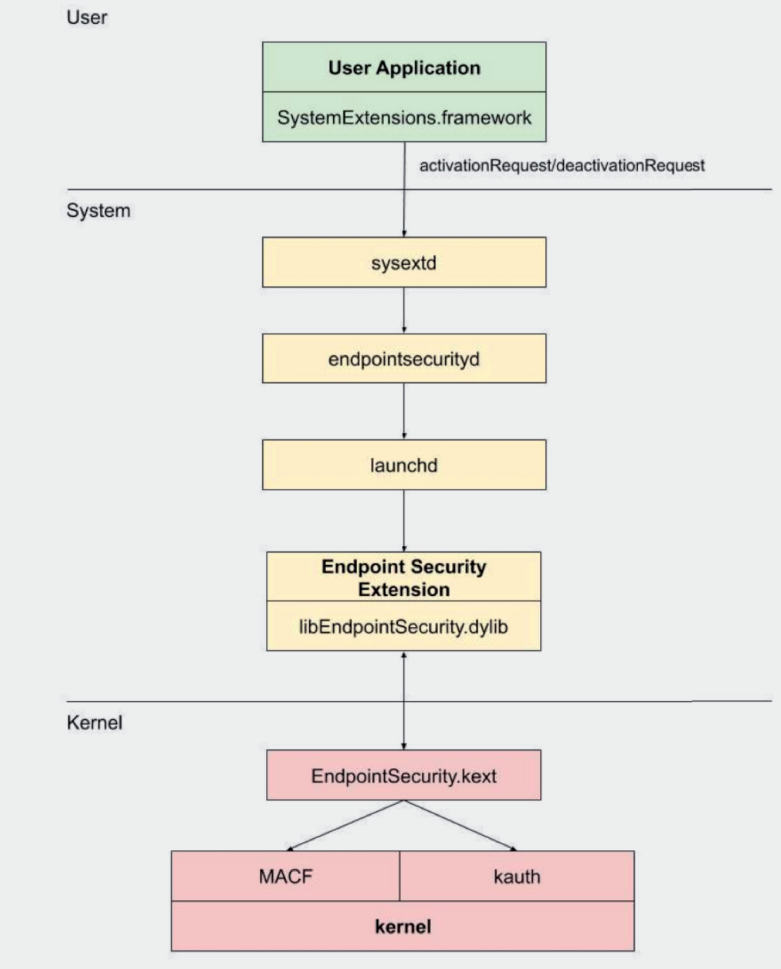

Figure 2: Framework architecture.

At the very bottom we have the `EndpointSecurity.kext` kernel extension. It integrates with the `MAC framework` and audit to accomplish all of its capabilities (the `kauth framework` integration was discontinued in *Big Sur*). At the time of this writing the Endpoint Security MAC policy module implements around 60 different MAC hooks. It is shown in Figure 3.

In user mode we have the `libEndppointSecurity.dylib`, which provides the C API for endpoint security clients. This is what we use when we create the actual ES client, and respond to ES events.

The `endpointsecurityd` daemon is responsible for loading the system extensions (SEXT) using launchd.

`sysextd` is responsible for validating the system extensions and if the code signature is valid, then it copies them to `/Library/SystemExtensions/`. This daemon will instruct `endpointsecurityd` to load the extension.

Finally, we have the `SystemExtension.framework`, which we can use from our application to activate or deactivate our system extension. Through this framework we can essentially call `sysextd` to initiate the load of the extension.

The `systemextensionsctl` command line utility provides basic control of the system extensions. We can list, reset and uninstall extensions using this tool.

The Endpoint Security framework supports various events, and it keeps increasing. These events are sent to our user-mode system extension, which allows our ES client to react to them, either passively through notification, or actively through authorization.

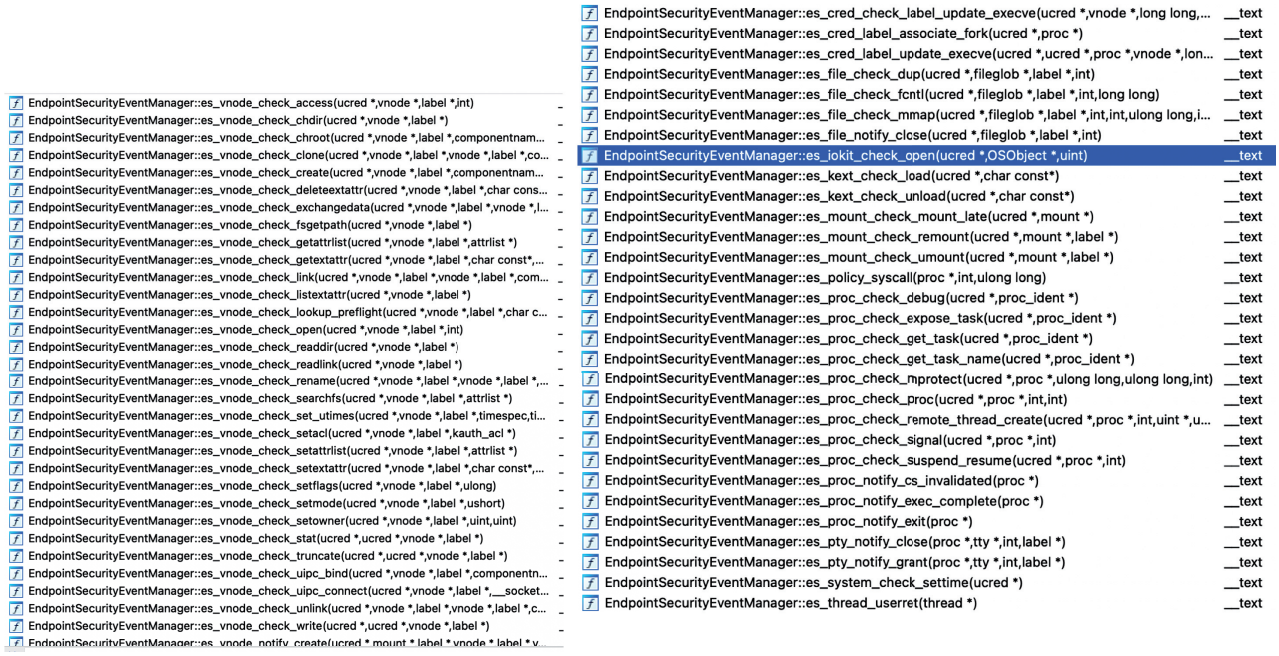

Figure 3: The Endpoint Security MAC policy module implements around 60 different MAC hooks.

Although the ES kernel extension is much more than a MAC policy module, on a very high level it extends the MAC hooks to user mode. For example, the following ES user-mode events are related to the following MAC hooks in kernel space.

ES user-mode event	MAC hook
ES_EVENT_TYPE_NOTIFY_CHROOT	es_vnode_check_chroot
ES_EVENT_TYPE_NOTIFY_MOUNT	es_mount_check_mount_late
ES_EVENT_TYPE_NOTIFY_MMAP	es_file_check_mmap
ES_EVENT_TYPE_AUTH_GET_TASK	es_proc_check_get_task

There is much more happening there, but essentially we got the MAC framework extended into user mode, which is very powerful. Considering that MAC was never officially supported by *Apple* for third-party kernel drivers, this is a huge thing.

The main drawback of the framework is that system extensions run in user mode, and have no possibility to inspect the kernel, thus if an adversary finds a way to compromise the kernel and install a kernel rootkit, we won't be able to detect it.

Next I will discuss how we can use develop our own Endpoint Security application and how we can use it to mitigate logic exploits.

EXPLOIT MITIGATION, SHIELD.APP

Getting entitled

If we want to develop an Endpoint Security system extension we need to request access to the `com.apple.developer.endpoint-security.client` entitlement, which is not available by default to all developers. Unfortunately, we depend on *Apple*'s goodwill to get access to this entitlement.

I went through a very disappointing process when I requested access to the entitlement.

I submitted my initial request in March 2020. In a month, *Apple* approved the use of the developer version of the entitlement, which allows a developer to install the system extension on their own Macs without the need for turning off SIP. This was a great step forward and meant I could start developing the application. In the meantime, I asked *Apple* to grant me access to the production version of the entitlement, which would allow me to distribute the application to users.

This is where it became an annoying and frustrating experience. *Apple* didn't reply to me apart from sending me one question in July 2020. It felt as if all of my emails were going into a black hole and no one was seeing them. I got so demotivated that I stopped developing my app for months, as I didn't see the point in writing it if no one would be able to use it except me.

Eventually, in January 2021, *Apple* granted me the entitlement. I don't know what happened or why I needed to wait so long. Luckily I don't do development for a living, and wasn't relying on this as a source of income – however for someone who does, this would be an even worse experience. *Apple* needs to improve its developer relations.

Writing an Endpoint Security client

As a start, I heavily relied on Patrick Wardle's open-source *Objective-See* [14] tools, especially the ProcessMonitor client, and I even used some classes from that application. I also checked Stephen Davis's Crescendo project [15]. Although it's written in Swift, I learned a lot from it as well. I'm really thankful for these open-source projects, and I can't emphasize enough how much I learned from them.

Let's review the very basics. The code snippet below shows the major items we need to implement when we create an ES client:

```
es_client_t* endpointClient = nil;

es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_EXEC, ...};

es_new_client(&endpointClient, ^(es_client_t *client, const es_message_t *message){
      //callback
      switch (message->event_type) {

      case ES_EVENT_TYPE_NOTIFY_EXEC:
       //do stuff
  });

es_subscribe(endpointClient, events, sizeof(events)/sizeof(events[0]))
```

First, we need to create a variable which will hold a reference to our ES client. It has a type of `es_client_t` and the variable is `endpointClient` in this example. Next, we create an array of `es_event_type_t`, where we define the various event types in which we are interested. A list of supported events can be found in *Apple Developer Documentation* [16].

Next, we create a new client using the `es_new_client` function, passing two arguments. The first is our `endpointClient` variable, while the second is a C block, which is essentially a function that will be called when an event occurs. This is where we can add our logic of what to do in the case of an event fire.

Finally, we need to call `es_subscribe`, where we can subscribe our ES client to our events of interest.

Patrick Wardle wrote a very detailed guide about how to write an Endpoint Security system extension, which can be found at [17] and [18].

Next I will discuss the logic behind the exploit mitigation.

The application's logic

Shield.app handles various events, and in the following section I will review each of them; what are they and how the application treats them. The app supports both monitoring and blocking mode, thus it's subscribed to authorization events where possible.

ES_EVENT_TYPE_AUTH_EXEC

This event fires upon new process execution. To prevent some of the code injection it will verify two things. First, it will check if any of the *Electron* application debug arguments are used. They are:

- --inspect
- --inspect-brk
- --remote-debugging-port

Additionally, it will verify if any environment variable that can be used for code injection is used when the process is launched. The ones it looks for are:

- DYLD_INSERT_LIBRARIES
- CFNETWORK_LIBRARY_PATH
- RAWCAMERA_BUNDLE_PATH
- ELECTRON_RUN_AS_NODE

If any of the above are found it will throw an alert or block the execution, depending on the settings.

ES_EVENT_TYPE_AUTH_GET_TASK

This event is triggered when an application tries to gain access to another application's task port. This action is simply denied or creates an alert without any further checks.

ES_EVENT_TYPE_AUTH_MMAP

This event is fired when a file is mapped to memory. I try to use this event to catch dylib hijacking by cross-checking the process's signature with the dylib being loaded. It tries to enforce library validation for dylib files. True library validation is done in-memory, mostly by AMFI. Unfortunately we can't perform the same from our ES client, and we have to rely on slower methods.

For the code signature verification I have to read the dylib from disk, which means lot of disk I/O, and thus it's a slower process, especially for larger applications like Xcode. I try to do caching here, but it can still be slow. Also, frameworks and other bundles are not monitored.

The following code snippet shows the requirement string I use for validation:

```
//set req string, teamid = of the process
//anchor apple = apple's own binary - safe
//anchor apple generic and certificate leaf [subject.CN] = \"Apple Mac OS Application Signing\" -
app store, assume safe
//anchor apple generic and certificate leaf[subject.OU] = \"%@\" - match dev teamid

NSString *requirementString = [NSString stringWithFormat:@"(anchor apple) or (anchor apple
generic and certificate leaf [subject.CN] = \"Apple Mac OS Application Signing\") or (anchor
apple generic and certificate leaf[subject.OU] = \"%@\")", process.teamID];
```

Apple binaries, *Mac App Store* binaries and matching Team ID are allowed to be mapped into the process.

ES_EVENT_TYPE_AUTH_LINK

This event happens when a hard link is created. The logic is shown below:

```
([file_uid intValue] != file.process.uid)  && !([file_uid intValue] > 0 && file.process.uid == 0)
```

The application checks whether the target location of the hard link is owned by the same user as the process creating the link, and if not it will check if the file has lower privileges than root and if the process runs as root. This tries to limit processes with lower privileges creating a link pointing to a location with higher privilege, because as we saw earlier it opens up the possibility for abuse.

ES_EVENT_TYPE_NOTIFY_CREATE

This event is triggered when a file is created. I use it to monitor for symbolic link creation. If the file is a symbolic link, the logic check is the same as in the case of hard links. Unfortunately, this event is only notification, as we can't do authorization in this case. This is because in the case of symlinks the target is not known until the link is actually created by the system.

CONCLUSION

On *macOS*, process injection and file link attacks lead to easily exploitable vulnerabilities. As these are logic bugs, they are more reliable to exploit than memory corruption bugs. On the other hand, the attacks have an easily detectable pattern, which can be detected or prevented by monitoring operating system behaviour. The Endpoint Security framework provided by *macOS* is a perfect solution for this as it offers us plenty of events, which we can use to monitor such activity. The richness of ES comes from the fact that it's closely integrated with the MAC framework in kernel space, which allows the inspection of over 200 system calls.

Although getting the entitlement for our application was a challenge, it's a great tool for making defensive security products. Its main drawback is that it has no visibility of kernel space and, as such, kernel compromise and kernel rootkits can't be detected. Hopefully *Apple* will strengthen this missing element in the future.

REFERENCES

[1] Fitzl, Cs. Shield - An app to protect against process injection on macOS. THEEVILBIT BLOG. https://theevilbit.github.io/shield/.

[2] https://github.com/theevilbit/Shield.

[3] MITRE ATT&CK. Process Injection. https://attack.mitre.org/techniques/T1055/.

[4] Fitzl, Cs. DYLD_INSERT_LIBRARIES DYLIB injection in macOS / OSX. THEEVILBIT BLOG. https://theevilbit.github.io/posts/dyld_insert_libraries_dylib_injection_in_macos_osx_deep_dive/.

[5] Fitzl, Cs. The Mysterious Syscall of AMFI. Offensive Security. https://www.offensive-security.com/offsec/amfi-syscall/.

[6] Wardle, P. Dylib hijacking on OS X. Virus Bulletin. https://www.virusbulletin.com/virusbulletin/2015/03/dylib-hijacking-os-x.

[7] Metnew, V. Why Electron apps can't store your secrets confidentially: ' — inspect' option. Medium. https://medium.com/@metnew/why-electron-apps-cant-store-your-secrets-confidentially-inspect-option-a49950d6d51f.

[8] Reguła, W. Abusing Electron apps to bypass macOS' security controls. https://wojciechregula.blog/post/abusing-electron-apps-to-bypass-macos-security-controls/.

[9] https://www.electronjs.org/apps.

[10] Fitzl, Cs. Exploiting directory permissions on macOS. THEEVILBIT BLOG. https://theevilbit.github.io/posts/exploiting_directory_permissions_on_macos/.

[11] Apple. Endpoint Security. https://developer.apple.com/documentation/endpointsecurity.

[12] Knight, S. System Extension internals. https://knight.sc/reverse%20engineering/2019/08/24/system-extension-internals.html.

[13] Knight, S. Endpoint Security and Insecurity. Objective by the Sea. https://objectivebythesea.com/v3/content.html.

[14] Objective-See. https://github.com/objective-see.

[15] Crescendo. https://github.com/SuprHackerSteve/Crescendo.

[16] Apple. es_event_type_t. https://developer.apple.com/documentation/endpointsecurity/es_event_type_t.

[17] Wardle, P. Writing a Process Monitor with Apple's Endpoint Security Framework. Objective-See. https://objective-see.com/blog/blog_0x47.html.

[18] Wardle, P. Writing a File Monitor with Apple's Endpoint Security Framework. Objective-See. https://objective-see.com/blog/blog_0x48.html.