Reverse Engineering the MacOS Sandbox

Hrant Tadevosyan

In contrast to Windows and Linux, MacOS by default, enforces kernel level sandboxing on executable binary files. This particular document will only focus on the aspect of so called 'entitlements'. The core implementation is done inside the XNU kernel, however, unlike the actual MacOS XNU Kernel [1], the source code is not available publicly. Both security researchers and regular software engineers can be interested in internal workings of the MacOS Sandboxing features. While the reason for security researchers is obvious, it is not quite clear why would a software engineer bother examining this component. Here are couple of points for it:

1. MacOS as opposed other operating systems such as Linux, Windows, OpenVMS and others, does not allow regular developers to use its internal user-space APIs. For example, in Windows, we can somewhat easily reverse engineer a utility tool which performs some unknown operations with the underlying system. This usually can be done via dynamic analysis (e.g.: debuggers). In MacOS one can find the necessary functions with dynamic analysis as well, however trying to use functions, coming out of the PrivateFrameworks, without the correct entitlements will cause the running executable to be killed immediately.

2. Knowing which functions can be used with the given developer certificates can be handy, since there is no generic way of identifying if a particular function will need specific rights or not.

3. The entitlements can be simply assigned to the executable, however, after running the executable binary file it might be forcibly closed. A software engineer most likely won't have a verbose error message telling why exactly it failed.

The goal of this paper is not to only give you an idea how the entitlements are tight to a running process, but also to give an insight how can a researcher analyze modern MacOS dynamic libraries, drivers or as commonly referred by Apple 'kernel extensions' and executables.

As an example we will pick the utility tool, named **tccutil**. It can be found in */usr/bin* folder (type **whereis tccutil** to confirm). This utility can be used to reset the Full Disk Access (FDA) rights from another executable. To enable it, regular MacOS users have to use the GUI interface. They have to go to *System Settings* > *Privacy and Security* > *Full Disk Access* and enable the entry listed there. On newer MacOS systems the **tccutil** can only be used to reset already granted FDA rights. Nevertheless, there is no documentation of a system API which is capable of doing it. Note that although we picked a very specific, all the steps will work for any other scenario.

First of all, we can start reverse engineering the utility tool. We can pull it into IDA [2] (Note that most of the labels were reverse engineered manually). From the following assembly listing we can see that it checks whether we supply the *reset* verb in arguments, if the comparison fails it simply prints the usage.

```
mov     rdi, [argv+8]    ; char *
lea     rsi, aReset      ; "reset"
call    _strcmp
test    eax, eax
jnz     print_usage
```

This means that we have to follow the other branch, which includes the following instructions (we also notice that the executable is fairly small which hints us that we'll probably have to look into library and/or drivers):

```
mov     rdi, cs:classRef_NSString ; id
mov     rcx, [argv+10h]
mov     rsi, cs:selRef_stringWithFormat_ ; SEL
lea     rdx, cstr_kTCCService%s
xor     eax, eax
call    cs:_objc_msgSend_ptr
mov     rdi, rax          ; id
call    _objc_retainAutoreleasedReturnValue
```

Due to the nature of higher level languages (compared to **C**) we are able to easily manually decompile the above listed instructions into the following approximate Objective-C code (consult the GNU LibObjC [3] reference implementation for details):

```
// for example the command: tccutil reset my_service
// will translate to: svc_str = "kTCCServicemy_service"
NSString *svc_str = [NSString stringWithFormat: @"kTCCService%s", argv[2]];
```

Later on we can see that is passed to a undocumented function, called *TCCAccessReset*. The return value is compared against zero. And if it is zero then it will print an error message:

```
mov     rdi, rbx          ; here is the 'kTCCService%s' string
call    _TCCAccessReset
test    al, al
jz      print_error
```

Looking into the imported functions, we can see that the function is coming from a library named **TCC.framework** which can be found at */System/Library/PrivateFrameworks*.

If this was Windows or Linux, then we would be done here. It only remains to link to this framework and use that function. Trying to do the same on MacOS yields a different result. Here is an example:

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

extern int TCCAccessReset(NSString *service);

int main(int argc, char *argv[])
{
        NSString *svc_str =
            [NSString stringWithFormat: @"kTCCService%s", argv[1]];

        int res = TCCAccessReset(svc_str);
        NSLog(@"Res: %i", res);

        return 0;
}
```

Compile the source file with this command and run it. Despite the very simple structure it fails to complete successfully (even with root privileges):

```
% clang -framework Foundation \
> -framework TCC -F /System/Library/PrivateFrameworks \
> <source>.m -o <program>

% ./<program> <service>
2024-10-07 08:46:08.410 <program>[5071:2670541] Res: 0

% sudo ./<program> <service>
2024-10-07 08:46:08.430 <program>[5071:2671239] Res: 0
```

Debugging this can be very hard. To identify what went wrong we have to first understand how that **TCC.framework** works. Ideally we'd like to reverse engineer that *TCCAccessReset* function. The framework file should be in */System/Library/PrivateFrameworks/TCC.framework/* folder, however, in modern MacOS systems they're stored in a **dyld** cache. The cache file can be found at */System/Volumes/Preboot/Cryptexes/OS/System/Library/dyld*:

```
% ls -l /System/Volumes/Preboot/Cryptexes/OS/System/Library/dyld
-rwxr-xr-x  1 root  admin  870088704 Dec 15  2023 dyld_shared_cache_x86_64h
-rwxr-xr-x  1 root  admin  803700736 Dec 15  2023 dyld_shared_cache_x86_64h.01
-rwxr-xr-x  1 root  admin  741998592 Dec 15  2023 dyld_shared_cache_x86_64h.02
-rwxr-xr-x  1 root  admin  729776128 Dec 15  2023 dyld_shared_cache_x86_64h.03
-rwxr-xr-x  1 root  admin  767868928 Dec 15  2023 dyld_shared_cache_x86_64h.04
-rwxr-xr-x  1 root  admin  218447872 Dec 15  2023 dyld_shared_cache_x86_64h.05
-rwxr-xr-x  1 root  admin     809894 Dec 15  2023 dyld_shared_cache_x86_64h.map
```

We can drag the *dyld_shared_cache_x86_64h* file into IDA. Increase the dependency depth to see the imported libraries as well. After IDA is done analyzing it, we can search for our candidate function. The function seems to be a wrapper for an internal function called *TCCResetInternal*

```
public _TCCAccessReset
_TCCAccessReset proc near
push    rbp
mov     rbp, rsp
mov     rsi, rdi
lea     rdi, aTccaccessreset_1 ; "TCCAccessResetInternal"
xor     edx, edx
xor     ecx, ecx
xor     r8d, r8d
pop     rbp
jmp     _TCCResetInternal
_TCCAccessReset endp
```

It passes the string *"TCCAccessResetInternal"* and the given first argument as the second argument, everything else is zeroed out. The equivalent C code would be:

```
int TCCAccessReset(NSString *svc_str)
{
        return TCCResetInternal("TCCAccessResetInternal", svc_str, 0, 0, 0);
}
```

The internal function itself is also very small. I removed small parts that are not relevant for this discussion:

```
_TCCResetInternal proc near
; ...
call    _tccd
; ...
call    _TCCResetInternalWithConnection
; ...
_TCCResetInternal endp
```

Eventually we get to these function calls. In oversimplified view, the *tccd* function does create an XPC connection with *"com.apple.tccd"* service. The comparison is there for the system version of tccd service. For now we'll ignore it:

```
_tccd proc near
; ...
lea     rax, aComAppleTccd ; "com.apple.tccd"
lea     r15, aComAppleTccdSy ; "com.apple.tccd.system"
cmovz   r15, rax
add     rbx, rbx
xor     edi, edi          ; priority
xor     esi, esi          ; flags
call    _dispatch_get_global_queue
mov     rdi, r15
mov     rsi, rax
mov     rdx, rbx
call    _xpc_connection_create_mach_service
; ...
_tccd endp
```

So this function gets the XPC connection where *TCCResetInternalWithConnection* is used to send the actual data. It creates an XPC message, copies the supplied service name, request id and other things which are ignored for our scenario, then sends it. Now, we know that there is an IPC communication present between **tccutil** and **tccd**. It's a common way to make these 'private' function calls on MacOS. Two things are interesting here:

1. Where is that **tccd** located? This can be done via **ps -ef | grep tccd** command (for example it can be at */System/Library/PrivateFrameworks/TCC.framework/Support/tccd*).
2. Can we find that request 'id' *TCCAccessResetInternal* in that service? This can be done by again, reverse engineering the service.

The first question was easy, for the second one we need to look in the strings section and find the cross references. The cross refernces show multiple results. The only relevant one is with *strcmp*. Where it checks if the given string contains *TCCAccessResetInternal* or not. If it does it calls its handler function:

```
; ...
lea     rsi, aTccaccessreset ; "TCCAccessResetInternal"
mov     rdi, r13        ; char *
call    _strcmp
test    eax, eax
jz      jmp_TCCAccessResetInternal
; ...
jmp_TCCAccessResetInternal:
mov     rdi, r12
mov     rsi, r15        ; id
call    _handle_TCCAccessResetInternal
```

To simplify this procedure, I'll only list the manually crafted call stack. One can manually disassemble all the listed entries to prove that they're correct. Here is the graph of function calls from up to bottom:

```
_handle_TCCAccessResetInternal
> [TCCDAttributionChain initWithMessage:]
>> [TCCDAttributionChain initWithMessage:evaluateResponsibility:]
>>> [TCCDAttributionChain initWithMessage:evaluateResponsibility:processInfo:]
>>>> [TCCDAttributionChain getAuditToken:fromMessage:]
>>>> [TCCDAttributionChain createProcessFromAuditToken:processInfo:]
>>>>> [TCCDProcess initWithAuditToken:responsibleIdentity:]
>>>>> ...
>>>>> Here it sets the entitelments. It gets the audit token and uses 'libsec'
>>>>> functions to copy the entitlements into its own data structure.
>>>>> All these functions can be found online and are semi-documented
>>>>> ...
>>>> [TCCDAttributionChain setRequestingProcess:]
>>>> [TCCDAttributionChain requestingProcess:]
...
>>>> [TCCDAttributionChain evaluateResponsibleProcess:]
>>>>> [TCCDAttributionChain accessingProcess:]
>>>>>> ...
>>>>>> [TCCDPlatform allTCCEntitlements:]
>>>>>> Another libsec function calls to verify entitlements
```

As we can see it gets the process audit token with system functions by supplying the pid. It then uses *SecCodeCopy** functions from *libsecurity* [4] to get the entitlements. Later on it check with *SecCodeCheckVerify** functions if they're valid or not.

Now, we know that there are lot of internal entitlement validation check present. We only need to figure out which ones do we need. We can check which one are allowed by analyzing the *[TCCDPlatform allTCCEntitlements]* method. The method itself is pretty much a wrapper for an internal function, which registers all the entitlements. The pseudo-code looks something like this:

```
TCCDPlatform_allTCCEntitlements()
{
        TCCDPlatform_allTCCEntitlements_internal();
}

TCCDPlatform_allTCCEntitlements_internal()
{
        entitlements[32] = {0};
        entitlements[0] = @"com.apple.private.tcc.manager";
        // ...
        entitlements[6] = @"com.apple.private.tcc.manager.access.read"
        // ...
        entitlements[8] = @"com.apple.private.tcc.manager.access.delete"
        // ...
}
```

To confirm our very long hypothesis, we can list out the entitlements that the **tccutil** has, by issuing *codesign -dv –entitlements - /usr/bin/tccutil* command:

```
% codesign -dv --entitlements - /usr/bin/tccutil
...
[Dict]
        [Key] com.apple.private.system-extensions.tcc
        [Value]
                [Bool] true
        [Key] com.apple.private.tcc.manager.access.delete
        [Value]
                [Array]
                        [String] kTCCServiceAll
        [Key] com.apple.private.tcc.manager.access.read
        [Value]
                [Array]
                        [String] kTCCServiceSystemPolicyAllFiles
```

There, we can see that the binary does indeed have the correct entitlements. This means we could use these for our custom example to enable the TCC operations. We ca create a temporary xml file which will include the following equivalent definitions:

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
        <dict>
                <key>com.apple.private.system-extensions.tcc</key>
                <true/>
                <key>com.apple.private.tcc.manager.access.delete</key>
                <array>
                        <string>kTCCServiceAll</string>
                </array>
                <key>com.apple.private.tcc.manager.access.read</key>
                <array>
                        <string>kTCCServiceSystemPolicyAllFiles</string>
                </array>
        </dict>
</plist>
```

After which we can simply issue the *codesign -s - -f –entitlements <xml_file> <program>* and run it again. Here are the results:

```
% codesign -s - -f --entitlements <xml_file> <program>
% ./<program> <service>
zsh: killed     sudo ./<program> <service>

% sudo dmesg | grep <program>
[519718.417587]: AMFI: '<path>/<program>' is adhoc signed.
[519718.418519]: AMFI: code signature validation failed.
[519718.418522]: AMFI: bailing out because of restricted entitlements.
[519718.418528]:
   mac_vnode_check_signature: <path>/<program>:
     code signature validation failed fatally: When validating <program>:
        Code has restricted entitlements,
        but the validation of its code signature failed.
        Unsatisfied Entitlements: ...
    validation of code signature failed through MACF policy: 1
[519718.418545]: check_signature[pid: 7197]: error = 1
[519718.418568]: proc 7197: load code signature error 4 for file "<program>"
```

The program was killed by the kernel as we can see in the system logs. The monitoring component seems to be the AMFI and the signature check was failed in the kernel function *mac_vnode_check_signature*. The generic function can be found in the XNU source files, however the actual implementation is hidden inside that AMFI driver. MacOS kernel driver can use the *mac_policy_ops* structure to assign a new vnode operations. To verify if the AMFI driver has done it or we have to, again, analyze it.

In MacOS the kernel extension can be found in */System/Library/Extensions*. The AMFI (Apple Mobile File Integrity) kext can be found at */System/Library/Extensions/AppleMobileFileIntegrity.kext*, however, as with PrivateFrameworks it does not contain the binary file in newer MacOS systems. Instead it is inside a cache file. The cache files for kernel extensions are located at */System/Library/KernelCollections*. Inside the *BootKernelExtensions.kc* kext cache we can find the AMFI driver. As with dyld cache, we can load this file into IDA and wait until its done analyzing it.

When its done, we can look for the kernel function *mac_policy_register*. It accepts among other the *mops* structure which contains the function pointers, specifically the one for *vnode_signature_check*. Looking into cross references we can find one interesting entry inside a function, called *initializeAppleMobileFileIntegrity*:

```
; ...
lea     rcx, vnode_check_signature
mov     [mops + vnode_check_signature_ptr], rcx
; ...
call    mac_policy_register
```

Another side note is that these functions are called when the kernel receives *execve* or *posix_spawn* syscall requests. Which precisely the operation we performed when we ran our program from the terminal.

Yet another interesting kernel function is the *IOTaskHasEntitlement* which can be seen in the XNU source codes everywhere. It is used to check if a particular process has the given entitlement or not. Inside it it calls the *queryEntitlementBooleanWithProc*. The implementation of this function is once again hidden inside the AMFI kext. The driver has to use *amfi_interface_register* publicly available function inside it to make the pointer to the AMFI functions available to the other kernel components.

Inside AMFI it calls *kauth_cred_proc_ref* to get the credentials structure. Then it calls *amfi_cslot_get* where it supplies the *cred->cr_label*. This in turn calls *mac_label_get* to obtain it.

The *vnode* structure contains the *cs_blobs* indirectly which is used for getting the signature.

```
struct ubc_info {
        // ...
        struct  cs_blob *cs_blobs;
        // ...
};

struct vnode {
        // ...
        union {
                        struct mount    *vu_mountedhere;
                        struct socket   *vu_socket;
                        struct specinfo *vu_specinfo;
                        struct fifoinfo *vu_fifoinfo;
                        struct ubc_info *vu_ubcinfo;
        } v_un;
        // ...
};
```

The fact that the information is tied to the *vnode*, we can assume that the signature checks only happen at the beginning, meaning if someone would change the executing at runtime, then no errors would occur. This is indeed the case, one can test this by applying process injection techniques to a valid Apple binary file.

References

[1] Apple, The XNU Kernel, GitHub.

[2] Hex-Rays, IDA64, Download.

[3] GNU, LibObjC, GitHub.

[4] Apple, LibSecurity, Manual.